目錄

Django 中文文档 1.8

译者：Django 文档协作翻译小组，来源：Django documentation，最后更新：2017.2.15。

Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

下载

- 在线阅读
- PDF下载
- EPUB下载
- MOBI下载
- 代码仓库

赞助我们





新手入门

初次接触 Django 或编程吗？ 从这里开始吧！

从零开始

Django 初探

由于Django是在一个快节奏的新闻编辑室环境下开发出来的，因此它被设计成让普通的网站开发工作简单而快 捷。以下简单介绍了如何用 Django 编写一个数据库驱动的Web应用程序。

本文档的目标是给你描述足够的技术细节能让你理解Django是如何工作的，但是它并不表示是一个新手指南或参考目录 – 其实这些我们都有! 当你准备新建一个项目，你可以 从新手指南开始 或者 深入阅读详细的文档.

设计你的模型(model)

尽管你在 Django 中可以不使用数据库，但是它提供了一个完善的可以用 Python 代码描述你的数据库结构的对象关联映射(ORM)。

数据模型语法 提供了许多丰富的方法来展现你的模型 – 到目前为止，它已经解决了两个多年积累下来数据库架构问题。下面是个简单的例子，可能被保存为 mysite/news/models.py:

```
class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __unicode__(self):
        return self.full_name

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)

    def __unicode__(self):
        return self.headline
```

安装它

下一步，运行 Django 命令行工具来自动创建数据库表：

```
manage.py syncdb
```

syncdb 命令会查找你所有可用的模型(models)然后在你的数据库中创建还不存在的数据库表。

享用便捷的 API

接着，你就可以使用一个便捷且功能丰富的 *Python API* 来访问你的数据。API 是动态生成的，不需要代码生成:

```
# 导入我们在 "news "应用中创建的模型。
>>> from news.models import Reporter, Article

# 在系统中还没有 reporters 。
>>> Reporter.objects.all()
[]

# 创建一个新的 Reporter 。
>>> r = Reporter(full_name='John Smith')

# 将对象保存到数据库。你需要显示的调用 save() 方法。
>>> r.save()

# 现在它拥有了一个ID。
>>> r.id
1

# 现在新的 reporter 已经存在数据库里了。
>>> Reporter.objects.all()
[<Reporter: John Smith>]

# 字段被表示为一个 Python 对象的属性。
>>> r.full_name
'John Smith'

# Django 提供了丰富的数据库查询 API。
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist. Lookup par
ameters were {'id': 2}

# 创建一个 article.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool'
,
...     content='Yeah.', reporter=r)
>>> a.save()

# 现在 article 已经存在数据库里了。
```

```
>>> Article.objects.all()
[<Article: Django is cool>]

# Article 对象有 API 可以访问到关联到 Reporter 对象。
>>> r = a.reporter
>>> r.full_name
'John Smith'

# 反之亦然：Reporter 对象也有访问 Article 对象的API。
>>> r.article_set.all()
[<Article: Django is cool>]

# API 会在幕后高效的关联表来满足你的关联查询的需求。
# 以下例子是找出名字开头为 "John" 的 reporter 的所有 articles 。
>>> Article.objects.filter(reporter__full_name__startswith="John
")
[<Article: Django is cool>]

# 通过更改一个对象的属性值，然后再调用 save() 方法来改变它。
>>> r.full_name = 'Billy Goat'
>>> r.save()

# 调用 delete() 方法来删除一个对象。
>>> r.delete()
```

一个动态的管理接口：它不仅仅是个脚手架 – 还是个完整的房子

一旦你的 models 被定义好，Django 能自动创建一个专业的，可以用于生产环境的管理界面 – 一个可让授权用户添加，修改和删除对象的网站。它使用起来非常简单只需在你的 admin site 中注册你的模型即可。:

```
# In models.py...

from django.db import models

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)


# In admin.py in the same directory...

import models
from django.contrib import admin

admin.site.register(models.Article)
```

这种设计理念是你的网站一般是由一个员工,或者客户，或者仅仅是你自己去编辑 – 而你应该不会想要仅仅为了管理内容而去创建后台界面。

在一个创建 Django 应用的典型工作流中，首先需要创建模型并尽可能快地启动和运行 admin sites， 让您的员工(或者客户)能够开始录入数据。然后,才开发展现数据给公众的方式。

设计你的 URLs

一个干净的，优雅的 URL 方案是一个高质量 Web 应用程序的重要细节。 Django 鼓励使用漂亮的 URL 设计，并且不鼓励把没必要的东西放到 URLs 里面，像 .php 或 .asp.

为了给一个 app 设计 URLs，你需要创建一个 Python 模块叫做 URLconf。这是一个你的 app 内容目录， 它包含一个简单的 URL 匹配模式与 Python 回调函数间的映射关系。这有助于解耦 Python 代码和 URLs 。

这是针对上面 Reporter/Article 例子所配置的 URLconf 大概样子:

```
from django.conf.urls import patterns

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive')
,
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_d
etail'),
)
```

上面的代码映射了 URLs ，从一个简单的正则表达式，到 Python 回调函数("views")所在的位置。 正则表达式通过圆括号来"捕获" URLs 中的值。当一个用户请求一个页面时， Django 将按照顺序去匹配每一个模式，并停在第一个匹配请求的 URL 上。(如果没有匹配到， Django 将会展示一个404的错误页面。) 整个过程是极快的，因为在加载时正则表达式就进行了编译。

一旦有一个正则表达式匹配上了，Django 将导入和调用对应的视图，它其实就是一个简单的 Python 函数。每个视图将得到一个 request 对象 – 它包含了 request 的 meta 信息 – 和正则表达式所捕获到的值。

例如：如果一个用户请求了个 URL "/articles/2005/05/39323/", Django 将会这样调用函数 news.views.article_detail(request, '2005', '05', '39323').

编写你的视图(views)

每个视图只负责两件事中的一件：返回一个包含请求页面内容的 HttpResponse 对象; 或抛出一个异常如 Http404 。至于其他就靠你了。

通常，一个视图会根据参数来检索数据，加载一个模板并且根据该模板来呈现检索出来的数据。 下面是个接上例的 year_archive 例子

```
def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    return render_to_response('news/year_archive.html', {'year':
 year, 'article_list': a_list})
```

这个例子使用了 Django 的 模板系统，该模板系统功能强大且简单易用，甚至非编程人员也会使用。

设计你的模板(templates)

上面的例子中载入了 news/year_archive.html 模板。

Django 有一个模板搜索路径板，它让你尽可能的减少冗余而重复利用模板。在你的 Django设置中，你可以指定一个查找模板的目录列表。如果一个模板没有在这个 列表中，那么它会去查找第二个，然后以此类推。

假设找到了模板 news/year_archive.html 。下面是它大概的样子:

```
{% extends "base.html" %}

{% block title %}Articles for {{ year }}{% endblock %}

{% block content %}
<h1>Articles for {{ year }}</h1>

{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

变量使用双花括号包围。`{{ article.headline }} 表示 “输出 article 的 headline 属性”。而点符号不仅用于表示属性查找，还可用于字典的键值查找、索引查找和函数调用。

注意 `{{ article.pub_date|date:"F j, Y" }}` 使用了 Unix 风格的“管道”(“|”符合)。这就是所谓的模板过滤器，一种通过变量来过滤值的方式。本例中，Python datetime 对象被过滤成指定的格式(在 PHP 的日期函数中可以见到这种变换)。

你可以无限制地串联使用多个过滤器。你可以编写自定义的过滤器。你可以定制自己的模板标记，在幕后运行自定义的 Python 代码。

最后，Django 使用了“模板继承”的概念：这就是 `{% extends "base.html" %}` 所做的事。它意味着 “首先载入名为 ‘base’ 的模板中的内容到当前模板，然后再处理本模板中的其余内容。”总之，模板继承让你在模板间大大减少冗余内容：每一个模板只需要定义它独特的部分即可。

下面是使用了 静态文件 的 “base.html” 模板的大概样子:

```
{% load staticfiles %}
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="{% static "images/sitelogo.png" %}" alt="Logo" />
    {% block content %}{% endblock %}
</body>
</html>
```

简单地说，它定义了网站的外观（含网站的 logo ），并留下了个“洞”让子模板来填充。这使站点的重新设计变得非常容易，只需改变一个文件 – “base.html” 模板。

它也可以让你创建一个网站的多个版本，不同的基础模板，而重用子模板。Django 的创建者已经利用这一技术来创造了显著不同的手机版本的网站 – 只需创建一个新的基础模板。

请注意，如果你喜欢其他模板系统，那么你可以不使用 Django 的模板系统。 虽然 Django 的模板系统特别集成了 Django 的模型层，但并没有强制你使用它。同理，你也可以不使用 Django 的数据库 API。您可以使用其他数据库抽象层，您可以读取 XML 文件，你可以从磁盘中读取文件，或任何你想要的方法去操作数据。 Django 的每个组成部分： 模型、视图和模板都可以解耦，以后会谈到。

这仅仅是一点皮毛

这里只是简要概述了 Django 的功能。以下是一些更有用的功能：

一个 缓存框架 可以与 memcached 或其他后端缓存集成。 一个 聚合框架 可以让创建 RSS 和 Atom 的 feeds 同写一个小小的 Python 类一样容易。 更性感的自动创建管理站点功能 – 本文仅仅触及了点皮毛。 显然，下一步你应该 下载 Django，阅读 入门教程 并且加入 社区. 感谢您的关注！

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

快速安装指南

在你开始使用 Django 之前，你需要先安装它。我们有一个 完整安装指南 它涵盖了所有的安装步骤和可能遇到的问题；本指南将会给你一个最简单、简洁的安装指引。

安装 **Python**

作为一个 Web 框架，Django 需要使用 Python 。它适用 2.6.5 到 2.7 的所有 Python 版本。它还具有 3.2 和 3.3 版本的实验性支持。所有这些 Python 版本都包含一个轻量级的数据库名叫 SQLite 。因此你现在还不需要建立一个数据库。

在 http://www.python.org 获取 Python 。如果你使用 Linux 或者 Mac OS X，那很可能已经安装了 Python 。

> **在 Jython 使用 Django**
>
> 如果你使用 *Jython* (一个在 Java 平台上实现的 Python)，你需要遵循一些额外的步骤。查看 在 *Jyton* 上运行 *Python* 获取详细信息。

在你的终端命令行(shell)下输入 python 来验证是否已经安装 Python ; 你将看到如下类似的提示信息:

```
Python 3.3.3 (default, Nov 26 2013, 13:33:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more inform
ation.
>>>
```

建立一个数据库

若你需要一个“大”数据库引擎，例如：PostgreSQL ，MySQL ，或 Oracle ，那此步骤是需要的。 想要安装这样一个数据库，请参考 数据库安装信息.

删除旧版本的 **Django**

如果你是从旧版本的 Django 升级安装，你将需要 在安装新版本之前先卸载旧版本的 *Django*.

安装 **Django**

你可以使用下面这简单的三个方式来安装 Django:

- 安装 你的操作系统所提供供的发行包 。对于操作系统提供了 Django 安装包的人来说，这是最快捷的安装方法。
- 安装官方正式发布的版本 。这是对于想要安装一个稳定版本而不介意运行一个稍旧版本的 Django 的人来说是最好的方式。
- 安装最新的开发版本 。这对于那些想要尝试最新最棒的特性而不担心运行崭新代码的用户来说是最好的。

> 总是参考你所使用的对应版本的 **Django** 文档！
>
> 如果采用了前两种方式进行安装，你需要注意在文档中标明在开发版中新增的标记。这个标记表明这个特性仅适用开发版的 Django ，而他们可能不在官方版本发布。

验证安装

为了验证 Django 被成功的安装到 Python 中，在你的终端命令行 (shell) 下输入 python 。 然后在 Python 提示符下，尝试导入 Django:

```
>>> import django
>>> print(django.get_version())
1.8
```

你可能已安装了其他版本的 Django 。

安装完成！

安装完成 – 现在你可以 学习入门教程.

> 译者：Django 文档协作翻译小组，原文：Installation。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

教程

编写你的第一个 Django 程序 第1部分

让我们通过例子来学习。

在本教程中，我们将引导您创建一个基本的投票应用。

它将包含两部分：

- 一个公共网站，可让人们查看投票的结果和让他们进行投票。
- 一个管理网站，可让你添加、修改和删除投票项目。

我们假设你已经 安装了 Django 。你可以运行以下命令来验证是否已经安装了 Django 和运行着的版本号：

```
python -c "import django; print(django.get_version())"
```

你应该看到你安装的 Django 版本或一个提示你 “No module named django” 的错误。此外，还应该检查下你的版本与本教程的版本是否一致。 若不一致，你可以参考 Django 版本对应的教程或者更新 Django 到最新版本。

请参考 如何安装 Django 中的意见先删除旧版本的 Django 再安装一个新的。

> 在哪里可以获得帮助：
>
> 如果您在学习本教程中遇到问题，请在 django-users 上发贴或者在 #django on irc.freenode.net 上与其他可能会帮助您的 Django 用户交流。

创建一个项目

如果这是你第一次使用 Django ，那么你必须进行一些初始设置。也就是通过自动生成代码来建立一个 Django 项目 project – 一个 Django 项目的设置集，包含了数据库配置、 Django 详细选项设置和应用特性配置。

在命令行中，使用 cd 命令进入你想存储代码所在的目录，然后运行以下命令：

```
django-admin.py startproject mysite
```

这将在当前目录创建一个 mysite 目录。如果失败了，请查看 Problems running django-admin.py.

> Note
>
> 你需要避免使用 python 保留字或 Django 组件名作为项目的名称。尤其是你应该避免使用的命名如： django (与 Django 本身会冲突) 或者 test (与 Python 内置的包名会冲突).
>
> 这段代码应该放在哪里？
>
> 如果你有一般 PHP 的编程背景（未使用流行的框架），可能会将你的代码放在 Web 服务器的文档根目录下（例如： `/var/www` ）。而在 Django 中，你不必这么做。将任何 Python 代码放在你的 Web 服务器文档根目录不会是一个好主意，因为这可能会增加人们通过 Web 方式查看到你的代码的风险。这不利于安全。

将你的代码放在你的文档根目录 以外 的某些目录, 例如 /home/mycode 。

让我们来看看 startproject 都创建了些什么:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

> 和你看到的不一样？
>
> 默认的项目布局最近刚刚改变过。如果你看到的是一个“扁平”结构的目录布局（没有内层 mysite/ 目录），你很可能正在使用一个和本教程版本不一致的 Django 版本。你需要切换到对应的旧版教程或者使用较新的 Django 版本。

这些文件是：

- 外层 mysite/ 目录只是你项目的一个容器。对于 Django 来说该目录名并不重要; 你可以重命名为你喜欢的。
- manage.py: 一个实用的命令行工具，可让你以各种方式与该 Django 项目进行交互。 你可以在 django-admin.py and manage.py 中查看关于 manage.py 所有的细节。
- 内层 mysite/ 目录是你项目中的实际 Python 包。该目录名就是 Python 包名，通过它你可以导入它里面的任何东西。 (e.g. import mysite.settings).
- mysite/**init**.py: 一个空文件，告诉 Python 该目录是一个 Python 包。(如果你是 Python 新手，请查看官方文档了解 关于包的更多内容 。)
- mysite/settings.py: 该 Django 项目的设置/配置。请查看 Django settings 将会告诉你如何设置。
- mysite/urls.py: 该 Django 项目的 URL 声明; 一份由 Django 驱动的网站“目录”。请查看 URL dispatcher 可以获取更多有关 URL 的信息。
- mysite/wsgi.py: 一个 WSGI 兼容的 Web 服务器的入口，以便运行你的项目。请查看 How to deploy with WSGI 获取更多细节。

开发用服务器

让我们来验证是否工作。从外层 mysite 目录切换进去，若准备好了就运行命令 `python manage.py runserver` 。你将会看到命令行输出如下内容：

```
Performing system checks...

0 errors found
May 13, 2015 - 15:50:53
Django version 1.8, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

你已经启动了 Django 开发服务器，一个纯粹的由 Python 编写的轻量级 Web 服务器。我们在 Django 内包含了这个服务器，这样你就可以迅速开发了，在产品投入使用之前不必去配置一台生产环境下的服务器 – 例如 Apache 。

现在是一个很好的提示时机：不要 在任何类似生产环境中使用此服务器。它仅适用于开发环境。(我们提供的是 Web 框架的业务，而不是 Web 服务器。)

现在服务器正在运行中，请在你的 Web 浏览器中访问 http://127.0.0.1:8000/ 。 你会看到一个令人愉悦的，柔和的淡蓝色 “Welcome to Django” 页面。它工作正常！

> 更改端口号
>
> 默认情况下，runserver 命令启动的开发服务器只监听本地 IP 的 8000 端口。
>
> 如果你想改变服务器的端口，把它作为一个命令行参数传递即可。例如以下命令启动的服务器将监听 8080 端口：
>
> ```
> python manage.py runserver 8080
> ```
>
> 如果你想改变服务器 IP ，把它和端口号一起传递即可。因此，要监听所有公共 IP 地址（如果你想在其他电脑上炫耀你的工作），请使用：
>
> ```
> python manage.py runserver 0.0.0.0:8000
> ```
>
> 有关开发服务器的完整文档可以在 runserver 内参考。

数据库设置

现在，编辑 mysite/settings.py 。 这是一个普通的 Python 模块，包含了代表 Django 设置的模块级变量。 更改 DATABASES 中 'default' 下的以下键的值，以匹配您的数据库连接设置。

- ENGINE – 从 'django.db.backends.postgresql_psycopg2', 'django.db.backends.mysql', 'django.db.backends.sqlite3', 'django.db.backends.oracle' 中选一个， 至于其他请查看 also available.
- NAME – 你的数据库名。如果你使用 SQLite，该数据库将是你计算机上的一个文件；在这种情况下，NAME 将是一个完整的绝对路径，而且还包含该文件的名称。如果该文件不存在，它会在第一次同步数据库时自动创建（见下文）。

当指定路径时，总是使用正斜杠，即使是在 Windows 下(例如： `C:/homes/user/mysite/sqlite3.db`) 。

- USER – 你的数据库用户名 (SQLite 下不需要) 。
- PASSWORD – 你的数据库密码 (SQLite 下不需要) 。
- HOST – 你的数据库主机地址。如果和你的数据库服务器是同一台物理机器，请将此处保留为空 (或者设置为 127.0.0.1) (SQLite 下不需要) 。查看 HOST 了解详细信息。

如果你是新建数据库，我们建议只使用 SQLite ，将 ENGINE 改为 'django.db.backends.sqlite3' 并且将 NAME 设置为你想存放数据库的地方。SQLite 是内置在 Python 中的，因此你不需要安装任何东西来支持你的数据库。

> Note
>
> 如果你使用 PostgreSQL 或者 MySQL，确保你已经创建了一个数据库。还是通过你的数据库交互接口中的 “CREATE DATABASE database_name;” 命令做到这一点的。 如果你使用 SQLite ，你不需要事先创建任何东西 - 在需要的时候，将会自动创建数据库文件。 当你编辑 settings.py 时，将 TIME_ZONE 修改为你所在的时区。默认值是美国中央时区（芝加哥）。

同时，注意文件底部的 INSTALLED_APPS 设置。它保存了当前 Django 实例已激活的所有 Django 应用。每个应用可以被多个项目使用，而且你可以打包和分发给其他人在他们的项目中使用。

默认情况下，INSTALLED_APPS 包含以下应用，这些都是由 Django 提供的：

- django.contrib.auth – 身份验证系统。
- django.contrib.contenttypes – 内容类型框架。
- django.contrib.sessions – session 框架。
- django.contrib.sites – 网站管理框架。
- django.contrib.messages – 消息框架。
- django.contrib.staticfiles – 静态文件管理框架。

这些应用在一般情况下是默认包含的。

所有这些应用中每个应用至少使用一个数据库表，所以在使用它们之前我们需要创建数据库中的表。要做到这一点，请运行以下命令：

```
python manage.py syncdb
```

syncdb 命令参照 INSTALLED_APPS 设置，并在你的 settings.py 文件所配置的数据库中创建必要的数据库表。每创建一个数据库表你都会看到一条消息，接着你会看到一个提示询问你是否想要在身份验证系统内创建个超级用户。按提示输入后结束。

如果你感兴趣，可以在你的数据库命令行下输入： `dt` (PostgreSQL), SHOW TABLES; (MySQL), 或 .schema (SQLite) 来列出 Django 所创建的表。

> 极简主义者
>
> 就像我们上面所说的，一般情况下以上应用都默认包含在内，但不是每个人都需要它们。如果不需要某些或全部应用，在运行 syncdb 命令前可从 INSTALLED_APPS 内随意注释或删除相应的行。syncdb 命令只会为 INSTALLED_APPS 内的应用创建表。

创建模型

现在你的项目开发环境建立好了， 你可以开工了。

你通过 Djaong 编写的每个应用都是由 Python 包组成的，这些包存放在你的 Python path 中并且遵循一定的命名规范。 Django 提供了个实用工具可以自动生成一个应用的基本目录架构，因此你可以专注于编写代码而不是去创建目录。

> 项目（Projects）vs. 应用（apps）
>
> 项目与应用之间有什么不同之处？应用是一个提供功能的 Web 应用 – 例如：一个博客系统、一个公共记录的数据库或者一个简单的投票系统。 项目是针对一个特定的 Web 网站相关的配置和其应用的组合。一个项目可以包含多个应用。一个应用可以在多个项目中使用。

你的应用可以存放在 Python path 中的任何位置。在本教材中，我们将通过你的 manage.py 文件创建我们的投票应用，以便它可以作为顶层模块导入，而不是作为 mysite 的子模块。

要创建你的应用，请确认与 manage.py 文件在同一的目录下并输入以下命令：

```
python manage.py startapp polls
```

这将创建一个 polls 目录，其展开的样子如下所示：:

```
polls/
    __init__.py
    models.py
    tests.py
    views.py
```

此目录结构就是投票应用。

在 Django 中编写一个有数据库支持的 Web 应用的第一步就是定义你的模型 – 从本质上讲就是数据库设计及其附加的元数据。

> 哲理
>
> 模型是有关你数据的唯一且明确的数据源。它包含了你所要存储的数据的基本字段和行为。 Django 遵循 DRY 原则 。目标是为了只在一个地方定义你的数据模型就可从中自动获取数据。

在这简单的投票应用中，我们将创建两个模型： Poll 和 Choice `。` Poll 有问题和发布日期两个字段。 `Choice` 有两个字段： 选项 (choice) 的文本内容和投票数。每一个 Choice 都与一个 Poll 关联。

这些概念都由简单的 Python 类来表现。编辑 polls/models.py 文件后如下所示：

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

代码很简单。每个模型都由继承自 django.db.models.Model 子类的类来描述。 每个模型都有一些类变量，每一个类变量都代表了一个数据库字段。

每个字段由一个 Field 的实例来表现 – 比如 CharField 表示字符类型的字段和 DateTimeField 表示日期时间型的字段。这会告诉 Django 每个字段都保存了什么类型的数据。

每一个 Field 实例的名字就是字段的名字（如： question 或者 pub_date ），其格式属于亲和机器式的。在你的 Python 的代码中会使用这个值，而你的数据库会将这个值作为表的列名。

你可以在初始化 Field 实例时使用第一个位置的可选参数来指定人类可读的名字。这在Django的内省部分中被使用到了，而且兼作文档的一部分来增强代码的可读性。若字段未提供该参数，Django 将使用符合机器习惯的名字。在本例中，我们仅定义了一个符合人类习惯的字段名 Poll.pub_date 。对于模型中的其他字段，机器名称就已经足够替代人类名称了。

一些 Field 实例是需要参数的。 例如 CharField 需要你指定 `~django.db.models.CharField.max_length` 。这不仅适用于数据库结构，以后我们还会看到也用于数据验证中。

一个 Field 实例可以有不同的可选参数； 在本例中，我们将 votes 的 default 的值设为 0 。

最后，注意我们使用了 ForeignKey 定义了一个关联。它告诉 Django 每一个 `Choice` 关联一个 Poll 。 Django 支持常见数据库的所有关联：多对一（ many-to-ones ），多对多（ many-to-manys ） 和 一对一 （ one-to-ones ）。

激活模型

刚才那点模型代码提供给 Django 大量信息。有了这些 Django 就可以做：

为该应用创建对应的数据库架构 (CREATE TABLE statements) 。 为 Poll 和 Choice 对象创建 Python 访问数据库的 API 。 但首先，我们需要告诉我们的项目已经安装了 polls 应用。

> 哲理
>
> Django 应用是“可插拔的”：你可以在多个项目使用一个应用，你还可以分发应用，因为它们没有被捆绑到一个给定的 Django 安装环境中。

再次编辑 settings.py 文件，在 INSTALLED_APPS 设置中加入 'polls' 字符。因此结果如下所示：

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
)
```

现在 Django 已经知道包含了 polls 应用。让我们运行如下命令：

```
python manage.py sql polls
```

你将看到类似如下所示内容 (有关投票应用的 CREATE TABLE SQL 语句) :

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id") DE
FERRABLE INITIALLY DEFERRED,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
COMMIT;
```

请注意如下事项：

- 确切的输出内容将取决于您使用的数据库会有所不同。
- 表名是自动生成的，通过组合应用名 (polls) 和小写的模型名 – poll 和 choice 。(你可以重写此行为。)
- 主键 (IDs) 是自动添加的。(你也可以重写此行为。)
- 按照惯例，Django 会在外键字段名上附加 "_id" 。 (是的，你仍然可以重写此行为。)
- 外键关系由 REFERENCES 语句显示声明。
- 生成 SQL 语句时针对你所使用的数据库，会为你自动处理特定于数据库的字段，例如 auto_increment (MySQL), serial (PostgreSQL), 或 or integer primary key (SQLite) 。 在引用字段名时也是如此 – 比如使用双引号或单引号。 本教材的作者所使用的是 PostgreSQL，因此例子中输出的是 PostgreSQL 的语法。
- 这些 sql 命令其实并没有在你的数据库中运行过 - 它只是在屏幕上显示出来，以便让你了解 Django 认为什么样的 SQL 是必须的。 如果你愿意，可以把 SQL 复制并粘帖到你的数据库命令行下去执行。 但是，我们很快就能看到，Django 提供了一个更简单的方法来执行此 SQL 。

如果你感兴趣，还可以运行以下命令：

- python manage.py validate – 检查在构建你的模型时是否有错误。
- python manage.py sqlcustom polls – 输出为应用定义的任何 custom SQL statements (例如表或约束的修改) 。
- python manage.py sqlclear polls – 根据存在于你的数据库中的表 (如果有的话) ，为应用输出必要的 DROP TABLE 。
- python manage.py sqlindexes polls – 为应用输出 CREATE INDEX 语句。
- python manage.py sqlall polls – 输出所有 SQL 语句：sql, sqlcustom, 和 sqlindexes 。

看看这些输出的命令可以帮助你理解框架底层实际上处理了些什么。

现在，再次运行 syncdb 命令在你的数据库中创建这些模型对应的表：

```
python manage.py syncdb
```

syncdb 命令会给在 INSTALLED_APPS 中有但数据库中没有对应表的应用执行 sqlall 操作。 该操作会为你上一次执行 syncdb 命令以来在项目中添加的任何应用创建对应的表、初始化数据和创建索引。 syncdb 命令只要你喜欢就可以任意调用，并且它仅会创建不存在的表。

请阅读 django-admin.py documentation 文档了解 manage.py 工具更多的功能。

玩转 API

现在，我们进入 Python 的交互式 shell 中玩弄 Django 提供给你的 API 。要调用 Python sell ，使用如下命令：

```
python manage.py shell
```

我们当前使用的环境不同于简单的输入 “python” 进入的 shell 环境，因为 manage.py 设置了 DJANGO_SETTINGS_MODULE 环境变量，该变量给定了 Django 需要导入的 settings.py 文件所在路径。

> 忽略 manage.py
>
> 若你不想使用 manage.py ，也是没有问题的。 仅需要将 DJANGO_SETTINGS_MODULE 环境变量值设为 mysite.settings 并在与 manage.py 文件所在同一目录下运行 python （ 或确保目录在 Python path 下，那 import mysite 就可以了 ）。
>
> 想了解更多的信息，请参考 django-admin.py 文档 。

一旦你进入了 shell，就可通过 database API 来浏览数据：:

```
>>> from polls.models import Poll, Choice   # Import the model c
lasses we just wrote.

#  系统中还没有 polls 。
>>> Poll.objects.all()
[]

# 创建一个新 Poll 。
# 在默认配置文件中时区支持配置是启用的，
# 因此 Django 希望为 pub_date 字段获取一个 datetime  with tzinfo 。
使用了 timezone.now()
# 而不是 datetime.datetime.now() 以便获取正确的值。
>>> from django.utils import timezone
>>> p = Poll(question="What's new?", pub_date=timezone.now())

# 保存对象到数据库中。你必须显示调用 save() 方法。
>>> p.save()

# 现在对象拥有了一个ID 。请注意这可能会显示 "1L" 而不是 "1"，取决于
# 你正在使用的数据库。 这没什么大不了的，它只是意味着你的数据库后端
# 喜欢返回的整型数作为 Python 的长整型对象而已。
>>> p.id
1

# 通过 Python 属性访问数据库中的列。
>>> p.question
"What's new?"
>>> p.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# 通过改为属性值来改变值，然后调用 save() 方法。
>>> p.question = "What's up?"
>>> p.save()

# objects.all() 用以显示数据库中所有的 polls 。
>>> Poll.objects.all()
[<Poll: Poll object>]
```

请稍等。 `<Poll: Poll object>` 这样显示对象绝对是无意义的。 让我们编辑 polls 模型（ 在 polls/models.py 文件中 ） 并且给 Poll 和 Choice 都添加一个 **unicode**() 方法来修正此错误：

```
class Poll(models.Model):
    # ...
    def __unicode__(self):
        return self.question

class Choice(models.Model):
    # ...
    def __unicode__(self):
        return self.choice_text
```

给你的模型添加 **unicode**() 方法是很重要的， 不仅是让你在命令行下有明确提示，而且在 Django 自动生成的管理界面中也会使用到对象的呈现。

> 为什么是 **unicode**() 而不是 **str**()?
>
> 如果你熟悉 Python，那么你可能会习惯在类中添加 **str**() 方法而不是 **unicode**() 方法。 We use 我们在这里使用 **unicode**() 是因为 Django 模型默认处理的是 Unicode 格式。当所有存储在数据库中的数据返回时都会转换为 Unicode 的格式。
>
> Django 模型有个默认的 **str**() 方法 会去调用 **unicode**() 并将结果转换为 UTF-8 编码的字符串。这就意味着 unicode(p) 会返回一个 Unicode 字符串，而 str(p) 会返回一个以 UTF-8 编码的普通字符串。
>
> 如果这让你感觉困惑，那么你只要记住在模型中添加 **unicode**() 方法。 运气好的话，这些代码会正常运行。

请注意这些都是普通的 Python 方法。让我们来添加个自定义方法，为了演示而已：

```
import datetime
from django.utils import timezone
# ...
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timede
lta(days=1)
```

请注意，增加了 import datetime 和 from django.utils import timezone, 是为了分别引用 Python 的标准库 datetime 模块和 Django 的 django.utils.timezone 中的 time-zone-related 实用工具 。如果你不熟悉在 Python 中处理时区，你可以在 时区支持文档 学到更多。

保存这些更改并且再次运行 python manage.py shell 以开启一个新的 Python shell:

```
>>> from polls.models import Poll, Choice

# 确认我们附加的  __unicode__() 正常运行。
```

```
>>> Poll.objects.all()
[<Poll: What's up?>]

# Django 提供了一个丰富的数据库查询 API ，
# 完全由关键字参数来驱动。
>>> Poll.objects.filter(id=1)
[<Poll: What's up?>]
>>> Poll.objects.filter(question__startswith='What')
[<Poll: What's up?>]

# 获取今年发起的投票。
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Poll.objects.get(pub_date__year=current_year)
<Poll: What's up?>

# 请求一个不存在的 ID ，这将引发一个异常。
>>> Poll.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Poll matching query does not exist. Lookup paramet
ers were {'id': 2}

# 根据主键查询是常见的情况，因此 Django 提供了一个
# 主键精确查找的快捷方式。
# 以下代码等同于 Poll.objects.get(id=1).
>>> Poll.objects.get(pk=1)
<Poll: What's up?>

# 确认我们自定义方法正常运行。
>>> p = Poll.objects.get(pk=1)
>>> p.was_published_recently()
True

# 给 Poll 设置一些 Choices 。通过 create 方法调用构造方法去创建一个新
# Choice 对象实例，执行 INSERT 语句后添加该 choice 到
# 可用的 choices 集中并返回这个新建的 Choice 对象实例。 Django 创建了
# 一个保存外键关联关系的集合 ( 例如 poll 的 choices) 以便可以通过 API
# 去访问。
>>> p = Poll.objects.get(pk=1)

# 从关联对象集中显示所有 choices  -- 到目前为止还没有。
>>> p.choice_set.all()
[]

# 创建三个 choices 。
>>> p.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice_text='Just hacking again', vo
tes=0)
```

```
# Choice 对象拥有访问它们关联的 Poll 对象的 API 。
>>> c.poll
<Poll: What's up?>

# 反之亦然： Poll 对象也可访问 Choice 对象。
>>> p.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking ag
ain>]
>>> p.choice_set.count()
3

# 只要你需要 API 会自动连续关联。
# 使用双下划线来隔离关联。
# 只要你想要几层关联就可以有几层关联，没有限制。
# 寻找和今年发起的任何 poll 有关的所有 Choices
# ( 重用我们在上面建立的 'current_year' 变量 )。
>>> Choice.objects.filter(poll__pub_date__year=current_year)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking ag
ain>]

# 让我们使用 delete() 删除 choices 中的一个。
>>> c = p.choice_set.filter(choice_text__startswith='Just hackin
g')
>>> c.delete()
```

欲了解更多有关模型关系的信息，请查看 访问关联对象 。欲了解更多有关如何使用双下划线来通过 API 执行字段查询的，请查看 字段查询 。 如需完整的数据库 API 信息，请查看我们的 数据库 API 参考 。

当你对 API 有所了解后, 请查看 教程 第2部分 来学习 Django 的自动生成的管理网站是如何工作的。

> 译者：Django 文档协作翻译小组，原文：Part 1: Models。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个 Django 程序 第2部分

本教程上接 教程 第1部分 。 我们将继续开发 Web-poll 应用，并且专注在 Django 的 自动生成的管理网站上。

> 哲理
>
> 为你的员工或客户生成添加、修改和删除内容的管理性网站是个单调乏味的工作。 出于这个原因，Django 根据模型完全自动化创建管理界面。
>
> Django 是在新闻编辑室环境下编写的，“内容发表者”和“公共”网站之间有 非常明显的界线。网站管理员使用这个系统来添加新闻、事件、体育成绩等等， 而这些内容会在公共网站上显示出来。Django 解决了为网站管理员创建统一 的管理界面用以编辑内容的问题。
>
> 管理界面不是让网站访问者使用的。它是为网站管理员准备的。

启用管理网站

- 默认情况下 Django 管理网站是不启用的 – 它是可选的。 要启用管理网站，需要做三件事：
- 在 INSTALLED_APPS 设置中取消 "django.contrib.admin" 的注释。
- 运行 python manage.py syncdb 命令。既然你添加了新应用到 INSTALLED_APPS 中，数据库表就需要更新。
- 编辑你的 mysite/urls.py 文件并且将有关管理的行取消注释 – 共有三行取消了注释。该文件是 URLconf ；我们将在下一个教程中深入探讨 URLconfs 。现在，你需要知道的是它将 URL 映射到应用。最后你拥有的 urls.py 文件看起来像这样:

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', '{{ project_name }}.views.home', name='home'),
    # url(r'^{{ project_name }}/', include('{{ project_name }}.f
oo.urls')),

    # Uncomment the admin/doc line below to enable admin documen
tation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls
')),

    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

(粗体显示的行就是那些需要取消注释的行。)

启动开发服务器

让我们启动开发服务器并浏览管理网站。

回想下教程的第一部分，像如下所示启动你的开发服务器：

```
python manage.py runserver
```

现在，打开一个浏览器并在本地域名上访问 “/admin/” – 例如 http://127.0.0.1:8000/admin/ 。你将看到管理员的登录界面：

Django administration

Username:

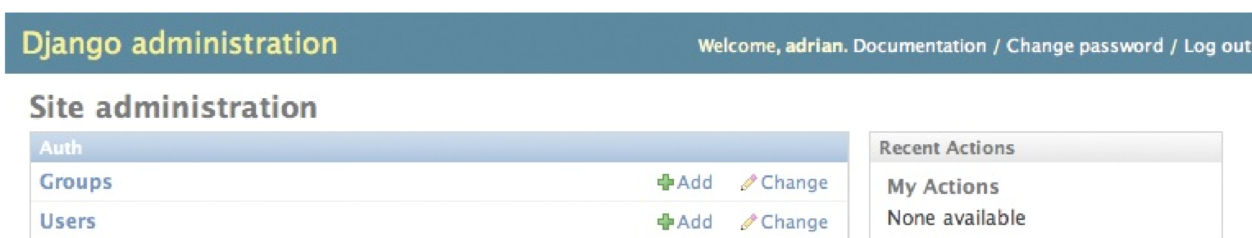
Password:

Log in

> 和你看到的不一样？
>
> 如果看到这，而不是上面的登录界面，那你应该得到一个类似如下所示的错误页面报告：
>
> ```
> ImportError at /admin/ cannot import name patterns ...
> ```
>
> 那么你很可能使用的 Django 版本不符合本教程的版本。 你可以切换到对应的旧版本教程去或者更新到较新的 Django 版本。

进入管理网站

现在尝试登录进去。（还记得吗？在本教程的第一部分时你创建过一个超级用户的帐号。如果你没有创建或忘记了密码，你可以 另外创建一个 。） 你将看到 Djaong 的管理索引页：



你将看到一些可编辑的内容，包括 groups ，users 和 sites 。这些都是 Django 默认情况下自带的核心功能。

使 poll 应用的数据在管理网站中可编辑

但是 poll 应用在哪？ 它可是没有在管理网站的首页上显示啊。

只需要做一件事：我们需要告诉管理网站 Poll 对象要有一个管理界面。为此，我们在你的 polls 目录下创建一个名为 admin.py 的文件，并添加如下内容：:

```
from django.contrib import admin
from polls.models import Poll
admin.site.register(Poll)
```

你需要重启开发服务器才能看到变化。通常情况下，你每次修改过一个文件后开发服务器都会自动载入，但是创建一个新文件却不会触发自动载入的逻辑。

探索管理功能

现在我们已经注册了 Poll ，那 Django 就知道了要在管理网站的首页上显示出来：

Site administration

Auth		
Groups	Add	Change
Users	Add	Change

Polls		
Questions	Add	Change

Recent Actions

My Actions
None available

点击 “Polls” 。现在你在 polls 的 “更改列表” 页。该页 显示了数据库中所有的 polls 可让你选中一个进行编辑。 有个 “What’s up?” poll 是我们在第一个教程中创建的：

Home › Polls › Questions

Select question to change

Add question

Action: --------- Go 0 of 1 selected

Question
What's up?

1 question

点击这个”What’s up?” 的 poll 进行编辑：

Home › Polls › Questions › What's up?

Change question

History

Question text: What's up?

Date published: Date: 2013-09-06 Today | Time: 16:42:32 Now |

Delete Save and add another Save and continue editing Save

这有些注意事项：

- 这的表单是根据 Poll 模型自动生成的。
- 不同模型的字段类型 (DateTimeField, CharField) 会对应的相应的 HTML 输入控件。 每一种类型的字段 Djaong 管理网站都知道如何显示它们。
- 每个 DateTimeField 都会有个方便的 JavaScript 快捷方式。日期有一个 “Today” 快捷方式和弹出式日历，而时间有个 “Now” 快捷方式和一个列出了常用时间选项的弹出式窗口。

在页面的底部还为你提供了几个选项：

- Save – 保存更改并返回到当前类型的对象的更改列表页面。
- Save and continue editing – 保存更改并重新载入当前对象的管理界面。
- Save and add another – 保存更改并载入当前对象类型的新的空白表单。
- Delete – 显示删除确认页。

如果 “Date published” 的值与你在第一部分教程时创建的 poll 的时间不符，这可能意味着你忘记了将 TIME_ZONE 设置成正确的值了。修改正确后再重启载入页面来检查值是否正确。

分别点击 “Today” 和 “Now” 快捷方式来修改 “Date published” 的值。 然后点击 “Save and continue editing” 。最后点击右上角的 “History” 。 你将看到一页列出了通过 Django 管理界面对此对象所做的全部更改的清单的页面， 包含有时间戳和修改人的姓名等信息：

Home › Polls › Questions › What's up? › History

Change history: What's up?

Date/time	User	Action
Sept. 6, 2013, 4:56 p.m.	rodolfo2488	Changed pub_date.

自定义管理表单

花些时间感叹一下吧，你没写什么代码就拥有了这一切。通过 admin.site.register(Poll) 注册了 Poll 模型，Django 就能构造一个默认的 表单。通常情况下，你将要自定义管理表单的外观和功能。这样的话你就需要在注册对象 时告诉 Django 对应的配置。

让我们来看看如何在编辑表单上给字段重新排序。将 admin.site.register(Poll) 这行替换成：:

```
class PollAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question']

admin.site.register(Poll, PollAdmin)
```

你将遵循这个模式 – 创建一个模型的管理对象，将它作为 admin.site.register() 方法的第二个参数传入 – 当你需要为一个对象做管理界面配置的时候。

上面那特定的更改使得 “Publication date” 字段在 “Question” 字段之前:

Home › Polls › Questions › What's up?

Change question

Date published: Date: 2013-09-03 Today | Time: 16:42:32 Now |

Question text: What's up?

Delete

仅有两个字段不会令你印象深刻，但是对于有许多字段的管理表单时，选择一个直观 的排序方式是一个重要的实用细节。

刚才所说的有许多字段的表单，你可能想将表单中的字段分割成 fieldsets ：:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Poll, PollAdmin)
```

在 fieldsets 中每一个 tuple 的第一个元素就是 fieldset 的标题。 下面是我们表单现在的样子：

你可以为每个 fieldset 指定 THML 样式类。Django 提供了一个 "collapse" 样式类用于显示初始时是收缩的 fieldset 。 当你有一个包含一些不常用的长窗体时这是非常有用的

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes':
 ['collapse']}),
    ]
Fieldset is initially collapsed
```

添加关联对象

Ok，现在我们有了 Poll 的管理页面。但是一个 Poll 拥有多个 Choices ，而 该管理页面并没有显示对应的 choices 。

是的。

我们有两种方法来解决这个问题。第一种就像刚才 Poll 那样在管理网站上 注册 Choice 。这很简单：

```
from polls.models import Choice

admin.site.register(Choice)
```

现在 “Choices” 在 Django 管理网站上是一个可用的选项了。”Add choice” 表单 看起来像这样：

该表单中，`Poll` 字段是一个包含了数据库中每个 poll 的选择框。 Django 知道 ForeignKey 在管理网站中以 `<select>` 框显示。在本例中，选择框中仅存在一个 poll 。

另外请注意 Poll 旁边的 “Add Another” 链接。每个有 ForeignKey 的对象关联到其他对象都会得到这个链接。 当点击 “Add Another” 时，你将会获得一个 “Add poll” 表单的弹出窗口。 如果你在窗口中添加了一 poll 并点击了 “Save” 按钮， Django 会将 poll 保存至数据库中并且动态的添加为你正在查看的 “Add choice” 表单中的已选择项。

但是，这真是一个低效的将 Choice 对象添加进系统的方式。 如果在创建 Poll 对象时能够直接添加一批 Choices 那会更好。 让我们这样做吧。

移除对 Choice 模型的 register() 方法调用 。然后，将 Poll 的注册代码 编辑为如下所示：

```
from django.contrib import admin
from polls.models import Choice, Poll

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes':
 ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

这将告诉 Django: “Choice 对象在 Poll 管理页面中被编辑。 默认情况下，提供 3 个 choices 的字段空间。

载入 “Add poll” 页面来看看，你可能需要重启你的开发服务器：

它看起来像这样：多了三个为关联 Choices 提供的输入插槽 – 由 extra 指定 – 并且每次你在 “Change” 页修改已经创建的对象时，都会另外获得三个额外插槽。

在现有的三个插槽的底部，你会发现一个 “Add another Choice” 链接。 如果你点击它，一个新的插槽会被添加。如果想移除添加的插槽， 你可以点击所添加的插槽的右上方的 X 。注意你不能移除原有的三个插槽。 此图片中显示了新增的插槽：

还有个小问题。为了显示所有关联 Choice 对象的字段需要占用大量的 屏幕空间。为此，Django 提供了一个以表格方式显示内嵌有关联对象的方式； 你只需要将 ChoiceInline 声明改为如下所示：

```
class ChoiceInline(admin.TabularInline):
    #...
```

使用了 TabularInline 后(而不是 StackedInline) ，基于表的格式下相关 对象被显示的更紧凑了：

需要注意的是有个额外的 “Delete?” 列允许保存时移除已保存过的行。

自定义管理界面的变更列表

现在 Poll 的管理界面看起来不错了，让我们给 “chang list” 页面做些调整 – 显示系统中所有 polls 的页面。

下面是现在的样子：

默认情况下， Django 显示的是每个对象 str() 的结果。但是若是我们能够 显示每个字段的话有时会更有帮助的。要做到这一点，需要使用 list_display 管理选项，这是一个 tuple ，包含了要显示的字段名， 将会以列的形式在该对象的 chang lsit 页上列出来：:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date')
```

效果再好的点话，让我们把在第一部分教程中自定义的方法 was_published_recently 也包括进来:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date', 'was_published_recen
tly')
```

现在 poll 的变更列表页看起来像这样：

Home › Polls › Questions

Select question to change

Action: --------- Go 0 of 1 selected

Question text	Date published	Was published recently
What's up?	Sept. 3, 2013, 4:42 p.m.	False

1 question

你可以点击列的标题对这些值进行排序 – 除了 was_published_recently 这一列，因为不支持根据方法输出的内容的排序。还要注意的是默认情况下列的标题是 was_published_recently ，就是方法名（将下划线替换为空格），并且每一行以字符串形式输出。

你可以通过给该方法 （在 models.py 内 ） 添加一些属性来改善显示效果，如下所示：:

```
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timede
lta(days=1)
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recent
ly?'
```

再次编辑你的 admin.py 文件并添加一个改进 Poll 的 change list 页面效果的功能：筛选 (Filters) 。在 PollAdmin 内添加一行如下所示的代码：:

```
list_filter = ['pub_date']
```

这就增加了一个“筛选” 的侧边栏，让人们通过 pub_date 字段的值来筛选 change list 显示的内容：

Home › Polls › Questions

Select question to change

Add question +

Action: --------- Go 0 of 1 selected

Question text	Date published	Published recently?
What's up?	Sept. 3, 2013, 4:42 p.m.	

1 question

Filter

By date published

Any date

Today

Past 7 days

This month

This year

显示筛选的类型取决于你需要筛选的字段类型。 因为 pub_date 是一个 DateTimeField 的实例，Django 知道提供对应的筛选选项：”Any date,” “Today,” “Past 7 days,” “This month,” “This year.”

为了效果更好。让我们来加上搜索功能：:

```
search_fields = ['question']
```

在 chang list 页的顶部增加了一个搜索框。当有人输入了搜索条件， Django 将搜索 question 字段。 虽然你可以使用任意数量的字段，如你希望的那样 – 但是因为它在后台用 LIKE 查询，为了保持数据库的性能请合理使用。

最后，因为 Poll 对象有日期字段，根据日期来向下钻取记录将会很方便。 添加下面这一行代码：:

```
date_hierarchy = 'pub_date'
```

这会在 change list 页的顶部增加了基于日期的分层导航功能。 在最顶层，显示所有可用年份。然后可钻取到月份，最终到天。

现在又是一个好时机，请注意 change lists 页面提供了分页功能。默认情况下每一页显示 100 条记录。 Change-list 分页，搜索框，筛选，日期分层和列标题排序如你所原地在一起运行了。

自定义管理界面的外观

显而易见，在每一个管理页面顶部有 “Django administration” 是无语的。虽然它仅仅是个占位符。

不过使用 Django 的模板系统是很容易改变的。Django 管理网站有 Django 框架自身的功能，可以通过 Django 自身的模板系统来修改界面。

自定义你的 项目 模板

在你的项目目录下创建一个 templates 目录。模板可以放在你的文件系统的任何地方，Diango 都能访问。 (Django 能以任何用户身份在你的服务器上运行。) 然后，在你的项目中保存模板是一个好习惯。

默认情况下，TEMPLATE_DIRS 值是空的。因此，让我们添加一行代码，来告诉 Django 我们的模板在哪里：:

```
TEMPLATE_DIRS = (
    '/path/to/mysite/templates', # 将此处改为你的目录。
)
```

现在从 Django 源代码中自带的默认 Django 管理模板的目录 (django/contrib/admin/templates) 下复制 admin/base_site.html 模板到你正在使用的 TEMPLATE_DIRS 中任何目录的子目录 admin 下。例如：如果你的 TEMPLATE_DIRS 中包含 '/path/to/mysite/templates' 目录， 如上所述，复制 django/contrib/admin/templates/admin/base_site.html 模板到 /path/to/mysite/templates/admin/base_site.html 。不要忘了是 admin 子目录。

> Django 的源代码在哪里？
>
> 如果在你的文件系统中很难找到 Django 源代码，可以运行如下命令：
>
> ```
> python -c "
> import sys
> sys.path = sys.path[1:]
> import django
> print(django.__path__)"
> ```

然后，只需要编辑该文件并将通用的 Djangot 文字替换为你认为适合的属于你自己的网站名。

该模板包含了大量的文字，比如 `{% block branding %}` 和 `{{ title }}` 。 `{%` 和 `{{` 标记是 Django 模板语言的一部分。 当 Django 呈现 admin/base_site.html 时，根据模板语言生成最终的 HTML 页面。 Don't worry if you can't make any sense of the template right now – 如果你现在不能理解模板的含义先不用担心 – 我们将在教程 3 中深入探讨 Django' 的模板语言。

请注意 Django 默认的管理网站中的任何模板都是可覆盖的。 要覆盖一个模板，只需要像刚才处理 base_site.html 一样 – 从默认的目录下复制到你的自定义目录下，并修改它。

自定义你的 应用 模板

细心的读者会问：如果 TEMPLATE_DIRS 默认的情况下是空值， 那 Django 是如何找到默认的管理网站的模板的？ 答案就是在默认情况下， Django 会自动在每一个应用的包内查找 templates/ 目录，作为备用使用。 （不要忘记 django.contrib.admin 是一个应用）。

我们的 poll 应用不是很复杂并不需要自定义管理模板。但是如果它变得更复杂 而且为了一些功能需要修改 Django 的标准管理模板，修改应用模板将是更 明智的选择，而不是修改项目模板。通过这种方式，你可以在任何新项目包括 polls 应用中自定义模板并且放心会找到需要的自定义的模板的。

有关 Django 怎样找到它的模板的更多信息，请参考 模板加载文档 。

自定义管理网站的首页

于此类似，你可能还想自定义 Django 管理网站的首页。

默认情况下，首页会显示在 INSTALLED_APPS 中所有注册了管理功能的应用，并按字母排序。你可能想在页面布局上做大修改。总之，首页可能是管理网站中最重要的页面， 因此它应该很容易使用。

你需要自定义的模板是 admin/index.html 。（同先前处理 admin/base_site.html 一样 – 从默认目录下复制到你自定义的模板目录下。） 编辑这个文件，你将看到一个名为 app_list 的模板变量。这个变量包含了每一个 已安装的 Django 应用。你可以通过你认为最好的方法硬编码链接到特定对象的管理页面，而不是使用默认模板。再次强调，如果你不能理解模板语言的话不用担心 – 我们将在教程 3 中详细介绍。

当你熟悉了管理网站的功能后，阅读 教程 第3部分 开始开发公共 poll 界面。

> 译者：Django 文档协作翻译小组，原文：Part 2: The admin site。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个 Django 程序 第3部分

本教程上接 教程 第2部分 。我们将继续 开发 Web-poll 应用并且专注在创建公共界面 – “视图 （views ）”。

哲理

在 Django 应用程序中，视图是一“类”具有特定功能和模板的网页。 例如，在一个博客应用程序中，你可能会有以下视图：

- 博客首页 – 显示最新发表的博客。
- 博客详细页面 – 一篇博客的独立页面。
- 基于年份的归档页 – 显示给定年份中发表博客的所有月份。
- 基于月份的归档页 – 显示给定月份中发表博客的所有日期。
- 基于日期的归档页 – 显示给定日期中发表的所有的博客。
- 评论功能 – 为一篇给定博客发表评论。

在我们的 poll 应用程序中，将有以下四个视图：

- Poll “index” 页 – 显示最新发布的民意调查。
- Poll “detail” 页 – 显示一项民意调查的具体问题，不显示该项的投票结果但可以进行投票的 form 。
- Poll “results” 页 – 显示一项给定的民意调查的投票结果。
- 投票功能 – 为一项给定的民意调查处理投票选项。

在 Django 中，网页及其他内容是由视图来展现的。而每个视图就是一个简单的 Python 函数（或方法， 对于基于类的视图情况下）。Django 会通过检查所请求的 URL （确切地说是域名之后的那部分 URL）来匹配一个视图。

平时你上网的时候可能会遇到像 “ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B” 这种如此美丽的 URL。 但是你会很高兴知道 Django 允许我们使用比那优雅的 URL 模式 来展现 URL。

URL 模式就是一个简单的一般形式的 URL - 比如: `/newsarchive/<year>/<month>/` .

Django 是通过 ‘URLconfs’ 从 URL 获取到视图的。而 URLconf 是将 URL 模式 (由正则表达式来描述的) 映射到视图的一种配置。

本教程中介绍了使用 URLconfs 的基本指令，你可以查阅 django.core.urlresolvers 来获取更多信息。

编写你的第一个视图

让我们编写第一个视图。打开文件 polls/views.py 并在其中输入以下 Python 代码

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the poll index.
")
```

在 Django 中这可能是最简单的视图了。为了调用这个视图，我们需要将它映射到一个 URL – 为此我们需要配置一个URLconf 。

在 polls 目录下创建一个名为 urls.py 的 URLconf 文档。 你的应用目录现在看起来像这样

```
polls/
    __init__.py
    admin.py
    models.py
    tests.py
    urls.py
    views.py
```

在 polls/urls.py 文件中输入以下代码：

```
from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',
    url(r'^$', views.index, name='index')
)
```

下一步是将 polls.urls 模块指向 root URLconf 。在 mysite/urls.py 中插入一个 include() 方法，最后的样子如下所示

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

现在你在 URLconf 中配置了 index 视图。通过浏览器访问 http://localhost:8000/polls/ ，如同你在 index 视图中定义的一样，你将看到 “Hello, world. You’re at the poll index.” 文字。

url() 函数有四个参数，两个必须的： regex 和 `view` ， 两个可选的： `kwargs` ， 以及 `name` 。 接下来，来探讨下这些参数的意义。

url() 参数: regex

regex 是 regular expression 的简写，这是字符串中的模式匹配的一种语法， 在 Django 中就是是 url 匹配模式。 Django 将请求的 URL 从上至下依次匹配列表中的正则表达式，直到匹配到一个为止。

需要注意的是，这些正则表达式不会匹配 GET 和 POST 参数，以及域名。 例如：针对 http://www.example.com/myapp/ 这一请求，URLconf 将只查找 myapp/ 。而在 http://www.example.com/myapp/?page=3 中 URLconf 也仅查找 myapp/ 。

如果你需要正则表达式方面的帮助，请参阅 Wikipedia's entry 和本文档中的 re 模块。 此外，O'Reilly 出版的由 Jeffrey Friedl 著的 "Mastering Regular Expressions" 也是不错的。 但是，实际上，你并不需要成为一个正则表达式的专家，仅仅需要知道如何捕获简单的模式。 事实上，复杂的正则表达式会降低查找性能，因此你不能完全依赖正则表达式的功能。

最后有个性能上的提示：这些正则表达式在 URLconf 模块第一次加载时会被编译。 因此它们速度超快 (像上面提到的那样只要查找的不是太复杂)。

url() 参数： view

当 Django 匹配了一个正则表达式就会调用指定的视图功能，包含一个 HttpRequest 实例作为第一个参数和正则表达式 "捕获" 的一些值的作为其他参数。 如果使用简单的正则捕获，将按顺序位置传参数；如果按命名的正则捕获，将按关键字传参数值。 有关这一点我们会给出一个例子。

url() 参数： kwargs

任意关键字参数可传一个字典至目标视图。在本教程中，我们并不打算使用 Django 这一特性。

url() 参数： name

命名你的 URL ，让你在 Django 的其他地方明确地引用它，特别是在模板中。 这一强大的功能可允许你通过一个文件就可全局修改项目中的 URL 模式。

编写更多视图

现在让我们添加一些视图到 polls/views.py 中去。这些视图与之前的略有不同，因为 它们有一个参数：:

```
def detail(request, poll_id):
    return HttpResponse("You're looking at poll %s." % poll_id)

def results(request, poll_id):
    return HttpResponse("You're looking at the results of poll %
s." % poll_id)

def vote(request, poll_id):
    return HttpResponse("You're voting on poll %s." % poll_id)
```

将新视图按如下所示的 url() 方法添加到 polls.urls 模块中去：:

```
from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<poll_id>\d+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<poll_id>\d+)/results/$', views.results, name='res
ults'),
    # ex: /polls/5/vote/
    url(r'^(?P<poll_id>\d+)/vote/$', views.vote, name='vote'),
)
```

在你的浏览器中访问 http://localhost:8000/polls/34/ 。将运行 detail() 方法并且显示你在 URL 中提供的任意 ID 。试着访问 http://localhost:8000/polls/34/results/ 和 http://localhost:8000/polls/34/vote/ – 将会显示对应的结果页及投票页。

当有人访问你的网站页面如 " /polls/34/ " 时，Django 会加载 mysite.urls 模块，这是因为 ROOT_URLCONF 设置指向它。接着在该模块中寻找名为 `urlpatterns` 的变量并依次匹配其中的正则表达式。 include() 可让我们便利地引用其他 URLconfs 。请注意 include() 中的正则表达式没有 $ (字符串结尾的匹配符 match character) 而尾部是一个反斜杠。当 Django 解析 include() 时，它截取匹配的 URL 那部分而把剩余的字符串交由 加载进来的 URLconf 作进一步处理。

include() 背后隐藏的想法是使 URLs 即插即用。 由于 polls 在自己的 URLconf(polls/urls.py) 中，因此它们可以被放置在 "/polls/" 路径下，或 "/fun_polls/" 路径下，或 "/content/polls/" 路径下，或者其他根路径，而应用仍可以运行。

以下是当用户访问 "/polls/34/" 路径时系统中将发生的事：

- Django 将寻找 '^polls/' 的匹配
- 接着，Django 截取匹配文本 ("polls/") 后剩余的文本 – "34/" – 传递到 'polls.urls' URLconf 中作进一步处理， 再将匹配 r'^(?P\d+)/$' 的结果作为参数传给 detail() 视图

```
detail(request=<HttpRequest object>, poll_id='34')
```

poll_id='34' 这部分就是来自 (?P\d+) 匹配的结果。 使用括号包围一个 正则表达式所"捕获"的文本可作为一个参数传给视图函数； `?P<poll_id>` 将会定义名称用于标识匹配的内容； 而 \d+ 是一个用于匹配数字序列（即一个数字）的正则表达式。

因为 URL 模式是正则表达式，所以你可以毫无限制地使用它们。但是不要加上 URL 多余的部分如 .html – 除非你想，那你可以像下面这样：:

```
(r'^polls/latest\.html$', 'polls.views.index'),
```

真的，不要这样做。这很傻。

在视图中添加些实际的功能

每个视图只负责以下两件事中的一件：返回一个 HttpResponse 对象，其中包含了所请求页面的内容， 或者抛出一个异常，例如 Http404 。剩下的就由你来实现了。

你的视图可以读取数据库记录，或者不用。它可以使用一个模板系统，例如 Django 的 – 或者第三方的 Python 模板系统 – 或不用。它可以生成一个 PDF 文件，输出 XML ， 即时创建 ZIP 文件， 你可以使用你想用的任何 Python 库来做你想做的任何事。

而 Django 只要求是一个 HttpResponse 或一个异常。

因为它很方便，那让我们来使用 Django 自己的数据库 API 吧， 在 教程 第1部分 中提过。修改下 index() 视图， 让它显示系统中最新发布的 5 个调查问题，以逗号分割并按发布日期排序：:

```
from django.http import HttpResponse

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.order_by('-pub_date')[:5]
    output = ', '.join([p.question for p in latest_poll_list])
    return HttpResponse(output)
```

在这就有了个问题，页面的设计是硬编码在视图中的。如果你想改变页面的外观，就必须修改这里的 Python 代码。因此，让我们使用 Django 的模板系统创建一个模板给视图用，就使页面设计从 Python 代码中 分离出来了。

首先，在 polls 目录下创建一个 templates 目录。 Django 将会在那寻找模板。

Django 的 TEMPLATE_LOADERS 配置中包含一个知道如何从各种来源导入模板的可调用的方法列表。 其中有一个默认值是 django.template.loaders.app_directories.Loader ，Django 就会在每个 INSTALLED_APPS 的 “templates” 子目录下查找模板 - 这就是 Django 知道怎么找到 polls 模板的原因，即使我们 没有修改 TEMPLATE_DIRS, 还是如同在 教程 第2部分 那样。

> 组织模板
>
> 我们 能够 在一个大的模板目录下一起共用我们所有的模板，而且它们会运行得很好。 但是，此模板属于 polls 应用，因此与我们在上一个教程中创建的管理模板不同， 我们要把这个模板放在应用的模板目录 (polls/templates) 下而不是项目的模板目录 (templates) 。 我们将在 可重用的应用教程 中详细讨论我们为什么 要这样做。

在你刚才创建的 `templates` 目录下，另外创建个名为 polls 的目录，并在其中创建一个 index.html 文件。换句话说，你的模板应该是 polls/templates/polls/index.html 。由于知道如上所述的 app_directories 模板加载器是 如何运行的，你可以参考 Django 内的模板简单的作为 polls/index.html 模板。

> 模板命名空间
>
> 现在我们 也许 能够直接把我们的模板放在 polls/templates 目录下 (而不是另外创建 polls 子目录)， 但它实际上是一个坏注意。 Django 将会选择第一个找到的按名称匹配的模板， 如果你在 不同 应用中有相同的名称的模板，Django 将无法区分它们。 我们想要让 Django 指向正确的模板，最简单的方法是通过 命名空间 来确保是 他们的模板。也就是说，将模板放在 另一个 目录下并命名为应用本身的名称。

将以下代码添加到该模板中:

```
{% if latest_poll_list %}
    <ul>
    {% for poll in latest_poll_list %}
        <li><a href="/polls/{{ poll.id }}/">{{ poll.question }}<
/a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

现在让我们在 index 视图中使用这个模板：

```
from django.http import HttpResponse
from django.template import Context, loader

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = Context({
        'latest_poll_list': latest_poll_list,
    })
    return HttpResponse(template.render(context))
```

代码将加载 polls/index.html 模板并传递一个 context 变量。 The context is a dictionary mapping template variable names to Python 该 context 变量是一个映射了 Python 对象到模板变量的字典。

在你的浏览器中加载 “/polls/” 页，你应该看到一个列表，包含了在教程 第1部分 中创建的 “What’s up” 调查。而链接指向 poll 的详细页面。

快捷方式: render()

这是一个非常常见的习惯用语，用于加载模板，填充上下文并返回一个含有模板渲染结果的 HttpResponse 对象。 Django 提供了一种快捷方式。这里重写完整的 index() 视图

```
from django.shortcuts import render

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[
:5]
    context = {'latest_poll_list': latest_poll_list}
    return render(request, 'polls/index.html', context)
```

请注意，一旦我们在所有视图中都这样做了，我们就不再需要导入 loader ， Context 和 HttpResponse (如果你仍然保留了 detail, `resutls` , 和 `vote` 方法，你还是需要保留 HttpResponse) 。

render() 函数中第一个参数是 request 对象，第二个参数是一个模板名称，第三个是一个字典类型的可选参数。 它将返回一个包含有给定模板根据给定的上下文渲染结果的 HttpResponse 对象。

抛出 404 异常

现在让我们解决 poll 的详细视图 – 该页显示一个给定 poll 的详细问题。 视图代码如下所示：:

```
from django.http import Http404
# ...
def detail(request, poll_id):
    try:
        poll = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render(request, 'polls/detail.html', {'poll': poll})
```

在这有个新概念：如果请求的 poll 的 ID 不存在，该视图将抛出 Http404 异常。

我们稍后讨论如何设置 polls/detail.html 模板，若是你想快速运行上面的例子， 在模板文件中添加如下代码：

```
{{ poll }}
```

现在你可以运行了。

快捷方式: get_object_or_404()

这很常见，当你使用 get() 获取对象时 对象却不存在时就会抛出 Http404 异常。对此 Django 提供了一个快捷操作。如下所示重写 detail() 视图：

```
from django.shortcuts import render, get_object_or_404
# ...
def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render(request, 'polls/detail.html', {'poll': poll})
```

get_object_or_404() 函数需要一个 Django 模型类作为第一个参数以及 一些关键字参数，它将这些参数传递给模型管理器中的 get() 函数。 若对象不存在时就抛出 Http404 异常。

> 哲理
>
> 为什么我们要使用一个 get_object_or_404() 辅助函数 而不是在更高级别自动捕获 ObjectDoesNotExist 异常， 或者由模型 API 抛出 Http404 异常而不是 ObjectDoesNotExist 异常？
>
> 因为那样会使模型层与视图层耦合在一起。Django 最重要的设计目标之一 就是保持松耦合。一些控制耦合在 django.shortcuts 模块中介绍。

还有个 get_list_or_404() 函数，与 get_object_or_404() 一样 – 不过执行的是 filter() 而不是 get() 。若返回的是空列表将抛出 Http404 异常。

编写一个 **404 (** 页面未找到 **)** 视图

当你在视图中抛出 Http404 时，Django 将载入一个特定的视图来处理 404 错误。Django 会根据你的 root URLconf (仅在你的 root URLconf 中；在其他任何地方设置 handler404 都无效 ）中设置的 handler404 变量来查找该视图，这个变量是个 Python 包格式字符串 – 和标准 URLconf 中的回调函数格式是一样的。 404 视图本身没有什么特殊性：它就是一个普通的视图。

通常你不必费心去编写 404 视图。若你没有设置 handler404 变量，默认情况下会使用内置的 django.views.defaults.page_not_found() 视图。或者你可以在你的模板目录下的根目录中 创建一个 404.html 模板。当 DEBUG 值是 False (在你的 settings 模块中) 时， 默认的 404 视图将使用此模板来显示所有的 404 错误。 如果你创建了这个模板，至少添加些如“页面未找到” 的内容。

一些有关 404 视图需要注意的事项 :

- 如果 DEBUG 设为 True (在你的 settings 模块里) 那么你的 404 视图将永远不会被使用 (因此 404.html 模板也将永远不会被渲染) 因为将要显示的是跟踪信息。
- 当 Django 在 URLconf 中不能找到能匹配的正则表达式时 404 视图也将被调用。 编写一个 500 (服务器错误) 视图

类似的，你可以在 root URLconf 中定义 handler500 变量，在服务器发生错误时 调用它指向的视图。服务器错误是指视图代码产生的运行时错误。

同样，你在模板根目录下创建一个 500.html 模板并且添加些像“出错了”的内容。

使用模板系统

回到我们 poll 应用的 detail() 视图中，指定 poll 变量后， `polls/detail.html` 模板可能看起来这样 :

```
<h1>{{ poll.question }}</h1>
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

模板系统使用了“变量.属性”的语法访问变量的属性值。 例如 `{{ poll.question }}` ， 首先 Django 对 poll 对象做字典查询。 否则 Django 会尝试属性查询 – 在本例中属性查询成功了。 如果属性查询还是失败了，Django 将尝试 list-index 查询。

在 `{% for %}` 循环中有方法调用: poll.choice_set.all 就是 Python 代码 poll.choice_set.all(),它将返回一组可迭代的 Choice 对象，可以用在 `{% for %}` 标签中。

请参阅 模板指南 来了解模板的更多内容。

移除模板中硬编码的 URLS

记得吗? 在 polls/index.html 模板中，我们链接到 poll 的链接是硬编码成这样子的：

```
<li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
```

问题出在硬编码，紧耦合使得在大量的模板中修改 URLs 成为富有挑战性的项目。不过，既然你在 polls.urls 模块中的 url() 函数中定义了 命名参数，那么就可以在 url 配置中使用 `{% url %}` 模板标记来移除特定的 URL 路径依赖:

```
<li><a href="{% url 'detail' poll.id %}">{{ poll.question }}</a>
</li>
```

> Note
>
> 如果 `{% url 'detail' poll.id %}` (含引号) 不能运行，但是 `{% url detail poll.id %}` (不含引号) 却能运行，那么意味着你使用的 Djang 低于 < 1.5 版。这样的话，你需要在模板文件的顶部添加如下的声明：:
>
> ```
> {% load url from future %}
> ```

其原理就是在 polls.urls 模块中寻找指定的 URL 定义。 你知道命名为 ‘detail’ 的 URL 就如下所示那样定义的一样：:

```
...
# 'name' 的值由 {% url %} 模板标记来引用
url(r'^(?P<poll_id>\d+)/$', views.detail, name='detail'),
...
```

如果你想将 polls 的 detail 视图的 URL 改成其他样子，或许像 polls/specifics/12/ 这样子，那就不需要在模板（或者模板集）中修改而只要在 polls/urls.py 修改就行了:

```
...
# 新增 'specifics'
url(r'^specifics/(?P<poll_id>\d+)/$', views.detail, name='detail
'),
...
```

URL 名称的命名空间

本教程中的项目只有一个应用： `polls` 。在实际的 Django 项目中，可能有 5、10、20 或者 更多的应用。Django 是如何区分它们的 URL 名称的呢？比如说， `polls` 应用有一个 detail 视图，而可能会在同一个项目中是一个博客应用的视图。Django 是如何知道 使用 `{% url %}` 模板标记创建应用的 url 时选择正确呢？

答案是在你的 root URLconf 配置中添加命名空间。在 mysite/urls.py 文件 (项目的 `urls.py` ，不是应用的) 中，修改为包含命名空间的定义:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls', namespace="polls")),
    url(r'^admin/', include(admin.site.urls)),
)
```

现在将你的 polls/index.html 模板中原来的 detail 视图:

```
<li><a href="{% url 'detail' poll.id %}">{{ poll.question }}</a>
</li>
```

修改为包含命名空间的 detail 视图:

```
<li><a href="{% url 'polls:detail' poll.id %}">{{ poll.question
}}</a></li>
```

当你编写视图熟练后，请阅读 教程 第4部分 来学习如何处理简单的表单和通用视图 。

> 译者：Django 文档协作翻译小组，原文：Part 3: Views and templates。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个 Django 程序 第4部分

本教程上接 教程 第3部分 。我们将 继续开发 Web-poll 应用并且关注在处理简单的窗体和优化我们的代码。

编写一个简单的窗体

让我们把在上一篇教程中编写的 poll 的 detail 模板更新下，在模板中包含 HTML 的组件:

```
<h1>{{ poll.question }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' poll.id %}" method="post">
{% csrf_token %}
{% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

简单的总结下:

- 上面的模板中为每个投票选项设置了一个单选按钮。每个单选按钮的 value 是投票选项对应的 ID 。每个单选按钮的 name 都是 `“choice”` 。这意味着，当有人选择了一个单选按钮并提交了表单，将会发送 的 POST 数据是 `choice=3` 。这是 HTML 表单中的基本概念。
- 我们将 form 的 action 设置为 `{% url 'polls:vote' poll.id %}，以及设置了` method="post"` 。使用 method="post" (而不是 method="get") 是非常重要的，因为这种提交表单的方式会改变服务器端的数据。 当你创建一个表单为了修改服务器端的数据时，请使用 method="post" 。这不是 Django 特定的技巧；这是优秀的 Web 开发实践。
- forloop.counter 表示 for 标签在循环中已经循环过的次数
- 由于我们要创建一个POST form (具有修改数据的功能)，我们需要担心跨站点请求伪造 （ Cross Site Request Forgeries ） 。 值得庆幸的是，你不必太担心这一点，因为 Django 自带了一个非常容易使用的系统来防御它。 总之，所有的 POST form 针对内部的 URLs 时都应该使用 `{% csrf_token %}` 模板标签。

现在，让我们来创建一个 Django 视图来处理提交的数据。 记得吗？在 教程 第3部分 中，我们为 polls 应用创建了一个 URLconf 配置中包含有这一行代码:

```
url(r'^(?P<poll_id>\d+)/vote/$', views.vote, name='vote'),
```

我们还创建了一个虚拟实现的 vote() 函数。让我们创建一个真实版本吧。在 polls/views.py 中添加如下代码:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
from polls.models import Choice, Poll
# ...
def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choi
ce'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the poll voting form.
        return render(request, 'polls/detail.html', {
            'poll': p,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successful
ly dealing
        # with POST data. This prevents data from being posted t
wice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', arg
s=(p.id,)))
```

在这代码中有些内容还未在本教程中提到过:

request.POST 是一个类似字典的对象，可以让你 通过关键字名称来获取提交的数据。在本例中， request.POST['choice'] 返回了所选择的投票项目的 ID ，以字符串的形式。 request.POST 的值永远是字符串形式的。

请注意 Django 也同样的提供了通过 request.GET 获取 GET 数据的方法 – 但是在代码中我们明确的使用了 request.POST 方法，以确保数据是通过 POST 方法来修改的。

如果 choice 未在 POST 数据中提供 request.POST['choice'] 将抛出 KeyError 当未给定 choice 对象时上面的代码若检测到抛出的是 KeyError 异常就会向 poll 显示一条错误信息。

在增加了投票选项的统计数后，代码返回一个 HttpResponseRedirect 对象而不是常见的 HttpResponse 对象。 HttpResponseRedirect 对象需要一个参数：用户将被重定向的 URL (请继续看下去在这情况下我们是如何构造 URL) 。

就像上面用 Python 作的注释那样，当成功的处理了 POST 数据后你应该总是返回一个 HttpResponseRedirect 对象。 这个技巧不是特定于 Django 的；它是优秀的 Web 开发实践。

在本例中，我们在 HttpResponseRedirect 的构造方法中使用了 reverse() 函数。此函数有助于避免在视图中硬编码 URL 的功能。它指定了我们想要的跳转的视图函数名以及视图函数中 URL 模式相应的可变参数。在本例中，我们使用了教程 第3部分中的 URLconf 配置， reverse() 将会返回类似如下所示的字符串

```
'/polls/3/results/'
```

... 在此 3 就是 p.id 的值。该重定向 URL 会调用 'results' 视图并显示最终页面。

正如在教程 第3部分提到的， `request` 是一个 HttpRequest 对象。想了解 HttpRequest 对象更多的内容，请参阅 request 和 response 文档 。

当有人投票后， `vote()` 视图会重定向到投票结果页。让我们来编写这个视图

```
def results(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render(request, 'polls/results.html', {'poll': poll})
```

这几乎和 教程 第3部分 中的 detail() 视图完全一样。 唯一的区别就是模板名称。稍后我们会解决这个冗余问题。

现在，创建一个 polls/results.html 模板:

```
<h1>{{ poll.question }}</h1>

<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ ch
oice.votes|pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' poll.id %}">Vote again?</a>
```

现在，在浏览器中访问 /polls/1/ 并完成投票。每次投票后你将会看到结果页数据都有更新。 如果你没有选择投票选项就提交了，将会看到错误的信息。

使用通用视图：优化代码

detail() (在 教程 第3部分 中) 和 results() 视图 都很简单 – 并且还有上面所提到的冗余问题。 `index()` 用于显示 polls 列表的 index() 视图 (也在教程 第3部分中)，也是存在类似的问题。

这些视图代表了基本的 Web 开发中一种常见的问题： 根据 URL 中的参数从数据库中获取数据，加载模板并返回渲染后的内容。由于这类现象很 常见，因此 Django 提供了一种快捷方式，被称之为“通用视图”系统。

通用视图抽象了常见的模式，以至于你不需要编写 Python 代码来编写一个应用。

让我们把 poll 应用修改成使用通用视图系统的应用，这样我们就能删除删除一些我们自己的代码了。 我们将采取以下步骤来进行修改：

- 修改 URLconf 。
- 删除一些旧的，不必要的视图。
- 修正 URL 处理到对应的新视图。

请继续阅读了解详细的信息。

> 为什么要重构代码？
>
> 通常情况下，当你编写一个 Django 应用时，你会评估下通用视图是否适合解决你的问题， 如果适合你就应该从一开始就使用它，而不是进行到一半才重构你的代码。 但是本教程直到现在都故意集中介绍“硬编码”视图，是为了专注于核心概念上。
>
> 就像你在使用计算器前需要知道基本的数学知识一样。

修改 URLconf

首先，打开 polls/urls.py 的 URLconf 配置文件并修改成如下所示样子

```
from django.conf.urls import patterns, url
from django.views.generic import DetailView, ListView
from polls.models import Poll

urlpatterns = patterns('',
    url(r'^$',
        ListView.as_view(
            queryset=Poll.objects.order_by('-pub_date')[:5],
            context_object_name='latest_poll_list',
            template_name='polls/index.html'),
        name='index'),
    url(r'^(?P<pk>\d+)/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/detail.html'),
        name='detail'),
    url(r'^(?P<pk>\d+)/results/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/results.html'),
        name='results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'polls.views.vote', name='v
ote'),
)
```

修改 views

在这我们将使用两个通用视图： ListView 和 DetailView 。这两个视图分别用于显示两种抽象概念 “显示一系列对象的列表” 和 “显示一个特定类型的对象的详细信息页”。

- 每个视图都需要知道使用哪个模型数据。因此需要提供将要使用的 model 参数。
- DetailView 通用视图期望从 URL 中捕获名为 "pk" 的主键值，因此我们将 poll_id 改为 pk 。

默认情况下， DetailView 通用视图使用名为 <应用名>/<模型名>_detail.html 的模板。在我们的例子中，将使用名为 "polls/poll_detail.html" 的模板。 template_name 参数是告诉 Django 使用指定的模板名，而不是使用自动生成的默认模板名。 我们也指定了 results 列表视图的 template_name – 这确保了 results 视图和 detail 视图渲染时会有不同的外观，虽然它们有一个 DetailView 隐藏在幕后。

同样的，~django.views.generic.list.ListView 通用视图使用的默认模板名为 <应用名>/<模型名>_list.html ；我们指定了 template_name 参数告诉 ListView 使用已经存在的 "polls/index.html" 模板。

在之前的教程中，模板提供的上下文中包含了 poll 和 latest_poll_list 上下文变量。在 DetailView 中 poll 变量是自动提供的 – 因为我们使用了一个 Django 模型 (Poll) ，Django 能够为上下文变量确定适合的名称。 另外 ListView 自动生成的上下文变量名是 poll_list 。若要覆盖此变量我们需要提供 context_object_name 选项， 我们想要使用 latest_poll_list 来替代它。作为一种替代方式，你可以改变你的模板来 匹配新的默认的上下文变量 – 但它是一个非常容易地告诉 Django 使用你想要的变量的方式。

现在你可以在 polls/views.py 中删除 index() ， detail() 和 results() 视图了。 我们不需要它们了 – 它们已替换为通用视图了。你也可以删除不再需要的 HttpResponse 导入包了。

运行服务器，并且使用下基于通用视图的新投票应用。

有关通用视图的完整详细信息，请参阅 通用视图文档.

当你熟悉了窗体和通用视图后，请阅读 教程 第5部分 来学习测试我们的投票应用。

> 译者：Django 文档协作翻译小组，原文：Part 4: Forms and generic views。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个Django应用，第5部分

本教程上接教程第4部分。 我们已经建立一个网页投票应用，现在我们将为它创建一些自动化测试。

自动化测试简介

什么是自动化测试？

测试是检查你的代码是否正常运行的简单程序。

测试可以划分为不同的级别。 一些测试可能专注于小细节（某一个模型的方法是否会返回预期的值？）， 其他的测试可能会检查软件的整体运行是否正常（用户在对网站进行了一系列的操作后，是否返回了正确的结果？）。这些其实和你早前在教程 1中做的差不多， 使用shell来检测一个方法的行为，或者运行程序并输入数据来检查它的行为方式。

自动化测试的不同之处就在于这些测试会由系统来帮你完成。你创建了一组测试程序，当你修改了你的应用，你就可以用这组测试程序来检查你的代码是否仍然同预期的那样运行，而无需执行耗时的手动测试。

为什么你需要创建测试

那么，为什么要创建测试？而且为什么是现在？

你可能感觉学习Python/Django已经足够，再去学习其他的东西也许需要付出巨大的努力而且没有必要。 毕竟，我们的投票应用已经活蹦乱跳了； 将时间运用在自动化测试上还不如运用在改进我们的应用上。 如果你学习Django就是为了创建一个投票应用，那么创建自动化测试显然没有必要。 但如果不是这样，现在是一个很好的学习机会。

测试将节省你的时间

在某种程度上，‘检查起来似乎正常工作’将是一种令人满意的测试。 在更复杂的应用中，你可能有几十种组件之间的复杂的相互作用。

这些组件的任何一个小的变化，都可能对应用的行为产生意想不到的影响。 检查起来‘似乎正常工作’可能意味着你需要运用二十种不同的测试数据来测试你代码的功能，仅仅是为了确保你没有搞砸某些事 —— 这不是对时间的有效利用。

尤其是当自动化测试只需要数秒就可以完成以上的任务时。 如果出现了错误，测试程序还能够帮助找出引发这个异常行为的代码。

有时候你可能会觉得编写测试程序将你从有价值的、创造性的编程工作里带出，带到了单调乏味、无趣的编写测试中，尤其是当你的代码工作正常时。

然而，比起用几个小时的时间来手动测试你的程序，或者试图找出代码中一个新引入的问题的原因，编写测试程序还是令人惬意的。

测试不仅仅可以发现问题，它们还能防止问题

将测试看做只是开发过程中消极的一面是错误的。

没有测试，应用的目的和意图将会变得相当模糊。 甚至在你查看自己的代码时，也不会发现这些代码真正干了些什么。

测试改变了这一切； 它们使你的代码内部变得明晰，当错误出现后，它们会明确地指出哪部分代码出了问题 —— 甚至你自己都不会料到问题会出现在那里。

测试使你的代码更受欢迎

你可能已经创建了一个堪称辉煌的软件，但是你会发现许多其他的开发者会由于它缺少测试程序而拒绝查看它一眼；没有测试程序，他们不会信任它。 Jacob Kaplan-Moss，Django最初的几个开发者之一，说过“不具有测试程序的代码是设计上的错误。”

你需要开始编写测试的另一个原因就是其他的开发者在他们认真研读你的代码前可能想要查看一下它有没有测试。

测试有助于团队合作

之前的观点是从单个开发人员来维护一个程序这个方向来阐述的。 复杂的应用将会被一个团队来维护。 测试能够减少同事在无意间破坏你的代码的机会（和你在不知情的情况下破坏别人的代码的机会）。 如果你想在团队中做一个好的Django开发者，你必须擅长测试！

基本的测试策略

编写测试有很多种方法。

一些开发者遵循一种叫做“由测试驱动的开发”的规则；他们在编写代码前会先编好测试。 这似乎与直觉不符，尽管这种方法与大多数人经常的做法很相似：人们先描述一个问题，然后创建一些代码来解决这个问题。 由测试驱动的开发可以用Python测试用例将这个问题简单地形式化。

更常见的情况是，刚接触测试的人会先编写一些代码，然后才决定为这些代码创建一些测试。 也许在之前就编写一些测试会好一点，但什么时候开始都不算晚。

有时候很难解决从什么地方开始编写测试。 如果你已经编写了数千行Python代码，挑选它们中的一些来进行测试不会是太容易的。 这种情况下，在下次你对代码进行变更，或者添加一个新功能或者修复一个bug时，编写你的第一个测试，效果会非常好。

现在，让我们马上来编写一个测试。

编写我们的第一个测试

我们找出一个错误

幸运的是，polls应用中有一个小错误让我们可以马上来修复它：如果Question在最后一个天发布，Question.was_published_recently() 方法返回True（这是对的），但是如果Question的pub_date 字段是在未来，它还返回True（这肯定是不对的）。

你可以在管理站点中看到这一点；创建一个发布时间在未来的一个Question；你可以看到Question 的变更列表声称它是最近发布的。

你还可以使用shell看到这点：

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the fu
ture
>>> future_question = Question(pub_date=timezone.now() + datetim
e.timedelta(days=30))
>>> # was it published recently
>>> future_question.was_published_recently()
True
```

由于将来的事情并不能称之为'最近'，这确实是一个错误。

创建一个测试来暴露这个错误

我们需要在自动化测试里做的和刚才在shell里做的差不多，让我们来将它转换成一个自动化测试。

应用的测试用例安装惯例一般放在该应用的tests.py文件中；测试系统将自动在任何以test开头的文件中查找测试用例。

将下面的代码放入polls应用下的tests.py文件中：

```
polls/tests.py
import datetime

from django.utils import timezone
from django.test import TestCase

from .models import Question


class QuestionMethodTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() should return False for questio
ns whose
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertEqual(future_question.was_published_recently(
), False)
```

我们在这里做的是创建一个django.test.TestCase子类，它具有一个方法可以创建一个pub_date在未来的Question实例。然后我们检查was_published_recently()的输出 —— 它应该是 False.

运行测试

在终端中，我们可以运行我们的测试：

```
$ python manage.py test polls
```

你将看到类似下面的输出：

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_question
    self.assertEqual(future_question.was_published_recently(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

发生了如下这些事：

- python manage.py test polls查找polls 应用下的测试用例
- 它找到 django.test.TestCase 类的一个子类
- 它为测试创建了一个特定的数据库
- 它查找用于测试的方法 —— 名字以test开始
- 它运行test_was_published_recently_with_future_question创建一个pub_date为未来30天的 Question实例
- ... 然后利用assertEqual()方法，它发现was_published_recently() 返回True，尽管我们希望它返回False

这个测试通知我们哪个测试失败，甚至是错误出现在哪一行。

修复这个错误

我们已经知道问题是什么：Question.was_published_recently() 应该返回 False，如果它的pub_date是在未来。在models.py中修复这个方法，让它只有当日期是在过去时才返回True ：

```
polls/models.py
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

再次运行测试：

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------
------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

在找出一个错误之后，我们编写一个测试来暴露这个错误，然后在代码中更正这个错误让我们的测试通过。

未来，我们的应用可能会出许多其它的错误，但是我们可以保证我们不会无意中再次引入这个错误，因为简单地运行一下这个测试就会立即提醒我们。 我们可以认为这个应用的这一小部分会永远安全了。

更加综合的测试

在这里，我们可以使was_published_recently() 方法更加稳定；事实上，在修复一个错误的时候引入一个新的错误将是一件很令人尴尬的事。

在同一个类中添加两个其它的测试方法，来更加综合地测试这个方法：

```
polls/tests.py
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() should return False for questions w
hose
    pub_date is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertEqual(old_question.was_published_recently(), Fals
e)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() should return True for questions wh
ose
    pub_date is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertEqual(recent_question.was_published_recently(), T
rue)
```

现在我们有三个测试来保证无论发布时间是在过去、现在还是未来Question.was_published_recently()都将返回合理的数据。

再说一次，polls 应用虽然简单，但是无论它今后会变得多么复杂以及会和多少其它的应用产生相互作用，我们都能保证我们刚刚为它编写过测试的那个方法会按照预期的那样工作。

测试一个视图

这个投票应用没有区分能力：它将会发布任何一个Question，包括 pub_date字段位于未来。我们应该改进这一点。 设定pub_date在未来应该表示Question在此刻发布，但是直到那个时间点才会变得可见。

视图的一个测试

当我们修复上面的错误时，我们先写测试，然后修改代码来修复它。 事实上，这是由测试驱动的开发的一个简单的例子，但做的顺序并不真的重要。

在我们的第一个测试中，我们专注于代码内部的行为。 在这个测试中，我们想要通过浏览器从用户的角度来检查它的行为。

在我们试着修复任何事情之前，让我们先查看一下我们能用到的工具。

Django测试客户端

Django提供了一个测试客户端来模拟用户和代码的交互。我们可以在tests.py 甚至在shell 中使用它。

我们将再次以shell开始，但是我们需要做很多在tests.py中不必做的事。首先是在shell中设置测试环境：

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

setup_test_environment()安装一个模板渲染器，可以使我们来检查响应的一些额外属性比如response.context，否则是访问不到的。请注意，这种方法不会建立一个测试数据库，所以以下命令将运行在现有的数据库上，输出的内容也会根据你已经创建的Question不同而稍有不同。

下一步我们需要导入测试客户端类（在之后的tests.py 中，我们将使用django.test.TestCase类，它具有自己的客户端，将不需要导入这个类）：

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

这些都做完之后，我们可以让这个客户端来为我们做一些事：

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/
polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
'\n\n\n    <p>No polls are available.</p>\n\n'
>>> # note - you might get unexpected results if your ``TIME_ZON
E``
>>> # in ``settings.py`` is not correct. If you need to change i
t,
>>> # you will also need to restart your shell session
>>> from polls.models import Question
>>> from django.utils import timezone
>>> # create a Question and save it
>>> q = Question(question_text="Who is your favorite Beatle?", p
ub_date=timezone.now())
>>> q.save()
>>> # check the response once again
>>> response = client.get('/polls/')
>>> response.content
'\n\n\n    <ul>\n    \n        <li><a href="/polls/1/">Who is yo
ur favorite Beatle?</a></li>\n    \n    </ul>\n\n'
>>> # If the following doesn't work, you probably omitted the ca
ll to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
[<Question: Who is your favorite Beatle?>]
```

改进我们的视图

投票的列表显示还没有发布的投票（即pub_date在未来的投票）。让我们来修复它。

在教程 4中，我们介绍了一个继承ListView的基于类的视图：

```
polls/views.py
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]
```

response.context_data['latest_question_list'] 取出由视图放置在context 中的数据。

我们需要修改get_queryset方法并让它将日期与timezone.now()进行比较。首先我们需要添加一行导入：

```
polls/views.py
from django.utils import timezone
```

然后我们必须像这样修改get_queryset方法：

```
polls/views.py
def get_queryset(self):
    """
    Return the last five published questions (not including thos
e set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter(pub_date__lte=timezone.now()) 返回一个查询集，包含pub_date小于等于timezone.now的Question。

测试我们的新视图

启动服务器、在浏览器中载入站点、创建一些发布时间在过去和将来的Questions，然后检验只有已经发布的Question会展示出来，现在你可以对自己感到满意了。你不想每次修改可能与这相关的代码时都重复这样做 —— 所以让我们基于以上shell会话中的内容，再编写一个测试。

将下面的代码添加到polls/tests.py：

```
polls/tests.py
from django.core.urlresolvers import reverse
```

我们将创建一个快捷函数来创建Question，同时我们要创建一个新的测试类：

```
polls/tests.py
def create_question(question_text, days):
    """
    Creates a question with the given `question_text` published
the given
    number of `days` offset to now (negative for questions publi
shed
    in the past, positive for questions that have yet to be publ
ished).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text,
                                   pub_date=time)


class QuestionViewTests(TestCase):
    def test_index_view_with_no_questions(self):
        """
        If no questions exist, an appropriate message should be
displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_questi
on_list'], [])

    def test_index_view_with_a_past_question(self):
        """
        Questions with a pub_date in the past should be displaye
d on the
        index page.
        """
        create_question(question_text="Past question.", days=-30
)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be di
splayed on
        the index page.
        """
        create_question(question_text="Future question.", days=3
0)
        response = self.client.get(reverse('polls:index'))
```

```
        self.assertContains(response, "No polls are available.",
                            status_code=200)
        self.assertQuerysetEqual(response.context['latest_questi
on_list'], [])

    def test_index_view_with_future_question_and_past_question(s
elf):
        """
        Even if both past and future questions exist, only past
questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30
)
        create_question(question_text="Future question.", days=3
0)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-
30)
        create_question(question_text="Past question 2.", days=-
5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>', '<Question: Past qu
estion 1.>']
        )
```

让我们更详细地看下以上这些内容。

第一个是Question的快捷函数create_question，将重复创建Question的过程封装在一起。

test_index_view_with_no_questions不创建任何Question，但会检查消息“No polls are available.” 并验证latest_question_list为空。注意django.test.TestCase类提供一些额外的断言方法。在这些例子中，我们使用assertContains() 和assertQuerysetEqual()。

在test_index_view_with_a_past_question中，我们创建一个Question并验证它是否出现在列表中。

在test_index_view_with_a_future_question中，我们创建一个pub_date 在未来的Question。数据库会为每一个测试方法进行重置，所以第一个Question已经不在那里，因此首页面里不应该有任何Question。

等等。 事实上，我们是在用测试模拟站点上的管理员输入和用户体验，检查针对系统每一个状态和状态的新变化，发布的是预期的结果。

测试 DetailView

一切都运行得很好； 然而，即使未来发布的Question不会出现在index中，如果用户知道或者猜出正确的URL依然可以访问它们。所以我们需要给DetailView添加一个这样的约束：

```
polls/views.py
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.no
w())
```

当然，我们将增加一些测试来检验pub_date 在过去的Question 可以显示出来，而pub_date在未来的不可以：

```
polls/tests.py
class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):
        """
        The detail view of a question with a pub_date in the fut
ure should
        return a 404 not found.
        """
        future_question = create_question(question_text='Future
question.',
                                          days=5)
        response = self.client.get(reverse('polls:detail',
                                   args=(future_question.id,)))
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_question(self):
        """
        The detail view of a question with a pub_date in the pas
t should
        display the question's text.
        """
        past_question = create_question(question_text='Past Ques
tion.',
                                        days=-5)
        response = self.client.get(reverse('polls:detail',
                                   args=(past_question.id,)))
        self.assertContains(response, past_question.question_tex
t,
                            status_code=200)
```

更多的测试思路

我们应该添加一个类似get_queryset的方法到ResultsView并为该视图创建一个新的类。这将与我们刚刚创建的非常类似；实际上将会有许多重复。

我们还可以在其它方面改进我们的应用，并随之不断增加测试。例如，发布一个没有Choices的Questions就显得傻傻的。所以，我们的视图应该检查这点并排除这些Questions。我们的测试应该创建一个不带Choices 的 Question然后测试它不会发布出来， 同时创建一个类似的带有 Choices的Question 并验证它会 发布出来。

也许登陆的用户应该被允许查看还没发布的 Questions，但普通游客不行。 再说一次：无论添加什么代码来完成这个要求，需要提供相应的测试代码，无论你是否是先编写测试然后让这些代码通过测试，还是先用代码解决其中的逻辑然后编写测试来证明它。

从某种程度上来说，你一定会查看你的测试，然后想知道是否你的测试程序过于臃肿，这将我们带向下面的内容：

测试越多越好

看起来我们的测试代码的增长正在失去控制。 以这样的速度，测试的代码量将很快超过我们的应用，对比我们其它优美简洁的代码，重复毫无美感。

没关系。让它们继续增长。最重要的是，你可以写一个测试一次，然后忘了它。 当你继续开发你的程序时，它将继续执行有用的功能。

有时，测试需要更新。 假设我们修改我们的视图使得只有具有Choices的Questions 才会发布。在这种情况下，我们许多已经存在的测试都将失败 —— 这会告诉我们哪些测试需要被修改来使得它们保持最新，所以从某种程度上讲，测试可以自己照顾自己。

在最坏的情况下，在你的开发过程中，你会发现许多测试现在变得冗余。 即使这样，也不是问题；对测试来说，冗余是一件好 事。

只要你的测试被合理地组织，它们就不会变得难以管理。 从经验上来说，好的做法是：

- 每个模型或视图具有一个单独的TestClass
- 为你想测试的每一种情况建立一个单独的测试方法
- 测试方法的名字可以描述它们的功能

进一步的测试

本教程只介绍了一些基本的测试。 还有很多你可以做，有许多非常有用的工具可以随便使用来你实现一些非常聪明的做法。

例如，虽然我们的测试覆盖了模型的内部逻辑和视图发布信息的方式，你可以使用一个“浏览器”框架例如Selenium来测试你的HTML文件在浏览器中真实渲染的样子。 这些工具不仅可以让你检查你的Django代码的行为，还能够检查你的JavaScript的行为。 它会启动一个浏览器，并开始与你的网站进行交互，就像有一个人在操纵一样，非常值得一看！ Django 包含一个LiveServerTestCase来帮助与Selenium 这样的工具集成。

如果你有一个复杂的应用，你可能为了实现continuous integration，想在每次提交代码后对代码进行自动化测试，让代码自动 —— 至少是部分自动 —— 地来控制它的质量。

发现你应用中未经测试的代码的一个好方法是检查测试代码的覆盖率。 这也有助于识别脆弱的甚至死代码。 如果你不能测试一段代码，这通常意味着这些代码需要被重构或者移除。 Coverage将帮助我们识别死代码。 查看与coverage.py 集成来了解更多细节。

Django 中的测试有关于测试更加全面的信息。

下一步？

关于测试的完整细节，请查看Django 中的测试。

当你对Django 视图的测试感到满意后，请阅读本教程的第6部分来 了解静态文件的管理。

> 译者：Django 文档协作翻译小组，原文：Part 5: Testing。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个Django应用，第6部分

本教程上接教程 5。 我们已经建立一个测试过的网页投票应用，现在我们将添加一张样式表和一张图片。

除了由服务器生成的HTML文件外，网页应用一般需要提供其它必要的文件 —— 比如图片文件、JavaScript脚本和CSS样式表 —— 来为用户呈现出一个完整的网站。在Django中，我们将这些文件称为“静态文件”。

对于小型项目，这不是个大问题，因为你可以将它们放在你的网页服务器可以访问到的地方。 然而，在大一点的项目中 —— 尤其是那些由多个应用组成的项目 —— 处理每个应用提供的多个静态文件集合开始变得很难。

这正是django.contrib.staticfiles的用途：它收集每个应用（和任何你指定的地方）的静态文件到一个单独的位置，这个位置在线上可以很容易维护。

自定义你的应用的外观

首先在你的polls中创建一个static目录。Django将在那里查找静态文件，与Django如何polls/templates/内部的模板类似。

Django 的 STATICFILES_FINDERS 设置包含一个查找器列表，它们知道如何从各种源找到静态文件。 其中默认的一个是AppDirectoriesFinder，它在每个INSTALLED_APPS下查找“static”子目录，就像刚刚在polls中创建的一样。管理站点也为它的静态文件使用相同的目录结构。

在你刚刚创建的static目录中，创建另外一个目录polls并在它下面创建一个文件style.css。换句话讲，你的样式表应该位于polls/static/polls/style.css。因为AppDirectoriesFinder 静态文件查找器的工作方式，你可以通过polls/style.css在Django中访问这个静态文件，与你如何访问模板的路径类似。

> 静态文件的命名空间
>
> 与模板类似，我们可以家那个我们的静态文件直接放在polls/static（而不是创建另外一个polls 子目录），但实际上这是一个坏主意。Django将使用它所找到的第一个文件名符合要求的静态文件，如果在你的不同应用中存在两个同名的静态文件，Django将无法区分它们。 我们需要告诉Django该使用其中的哪一个，最简单的方法就是为它们添加命名空间。 也就是说，将这些静态文件放进以它们所在的应用的名字命名的另外一个目录下。

将下面的代码放入样式表中 (polls/static/polls/style.css)：

```
polls/static/polls/style.css
li a {
    color: green;
}
```

下一步，在polls/templates/polls/index.html的顶端添加如下内容 ：

```
polls/templates/polls/index.html
{% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/st
yle.css' %}" />
```

`{% load staticfiles %}` 从staticfiles模板库加载 `{% static %}` 模板标签。 `{% static %}` 模板标签会生成静态文件的绝对URL。

这就是你在开发过程中，所需要对静态文件做的所有处理。 重新加载 http://localhost:8000/polls/ ，你应该会看到Question的超链接变成了绿色（Django的风格！），这意味着你的样式表被成功导入。

添加一张背景图片

下一步，我们将创建一个子目录来存放图片。 在polls/static/polls/目录中创建一个images 子目录。在这个目录中，放入一张图片background.gif。换句话，将你的图片放在 polls/static/polls/images/background.gif。

然后，向你的样式表添加（polls/static/polls/style.css）：

```
polls/static/polls/style.css
body {
    background: white url("images/background.gif") no-repeat rig
ht bottom;
}
```

重新加载 http://localhost:8000/polls/ ，你应该在屏幕的右下方看到载入的背景图片。

> 警告：
>
> 当然， `{% static %}` 模板标签不能用在静态文件（比如样式表）中，因为他们不是由Django生成的。 你应该永远使用相对路径来相互链接静态文件，因为这样你可以改变STATIC_URL （ static模板标签用它来生成URLs）而不用同时修改一大堆静态文件的路径。

这些知识基础。关于静态文件设置的更多细节和框架中包含的其它部分，参见静态文件 howto 和静态文件参考。部署静态文件讨论如何在真实的服务器上使用静态文件。

下一步？

新手教程到此结束。 在这期间，你可能想要在如何查看文档中了解文档的结构和查找相关信息方法。

如果你熟悉Python 打包的技术，并且对如何将投票应用制作成一个“可重用的应用”感兴趣，请看高级教程：如何编写可重用的应用。

> 译者：Django 文档协作翻译小组，原文：Part 6: Static files。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

高级教程

高级教程：如何编写可重用的应用

本高级教程上接教程 6。我们将把我们的网页投票转换成一个独立的Python包，这样你可以在其它项目中重用或者分享给其它人。

如果你最近没有完成教程1–6，我们建议你阅读它们使得你的示例项目与下面描述的相匹配。

可重用很重要

设计、构建、测试和维护一个网页应用有许多工作要做。许多Python 和 Django 项目都有常见的共同问题。如果我们可以节省一些这些重复的工作会不会很棒？

可重用性是Python 中一种生活的态度。Python包索引 (PyPI) 具有广泛的包，你可以在你自己的Python程序中使用。调查一下Django Packages中已经存在的可重用的应用，你可以结合它们到你的项目。Django 自身也只是一个Python 包。这意味着你可以获取已经存在的Python包和Django应用并将它们融合到你自己的网页项目。你只需要编写你项目的独特的部分。

比如说，你正在开始一个新的项目，需要一个像我们正在编写的投票应用。你如何让该应用可重用？幸运的是，你已经在正确的道路上。在教程 3中，我们看到我们可以如何使用include将投票应用从项目级别的URLconf 解耦。在本教程中，我们将更进一步，让你的应用在新的项目中容易地使用并随时可以发布给其它人安装和使用。

> 包？应用？
>
> Python 包 提供的方式是分组相关的Python 代码以容易地重用。一个包包含一个或多个Python代码（也叫做“模块”）。
>
> 包可以通过import foo.bar 或from foo import bar 导入。如果一个目录（例如polls）想要形成一个包，它必须包含一个特殊的文件**init**.py，即使这个文件为空。
>
> 一个Django 应用 只是一个Python包，它特意用于Django项目中。一个应用可以使用常见的Django 约定，例如具有models、tests、urls和views 子模块。
>
> 后面我们使用打包这个词来描述将一个Python包变得让其他人易于安装的过程。我们知道，这可能有点绕人。

你的项目和你的可重用的应用

经过前面的教程之后，我们的项目应该看上去像这样：

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    polls/
        __init__.py
        admin.py
        migrations/
            __init__.py
            0001_initial.py
        models.py
        static/
            polls/
                images/
                    background.gif
                style.css
        templates/
            polls/
                detail.html
                index.html
                results.html
        tests.py
        urls.py
        views.py
    templates/
        admin/
            base_site.html
```

你在教程 2中创建了mysite/templates ，在教程 3中创建了polls/templates。 现在你可能更加清晰为什么我们为项目和应用选择单独的模板目录：属于投票应用的部分全部在polls中。它使得该应用自包含且更加容易丢到一个新的项目中。

现在可以拷贝polls目录到一个新的Django项目并立即使用。然后它还不能充分准备好到可以立即发布。由于这点，我们需要打包这个应用来让它对其他人易于安装。

安装一些前提条件

Python 打包的目前状态因为有多种工具而混乱不堪。对于本教程，我们打算使用setuptools来构建我们的包。它是推荐的打包工具（已经与distribute 分支合并）。我们还将使用pip来安装和卸载它。现在你应该安装这两个包。如果你需要帮助，你可以参考如何使用pip安装Django。你可以使用同样的方法安装setuptools。

打包你的应用

Python packaging refers to preparing your app in a specific format that can be easily installed and used. Django 自己是以非常相似的方式打包起来的。对于一个像polls这样的小应用，这个过程不是太难。

首先，在你的Django项目之外，为polls创建一个父目录。称这个目录为django-polls。

> 为你的应用选择一个名字
>
> 让为你的包选择一个名字时，检查一下PyPI中的资源以避免与已经存在的包有名字冲突。当创建一个要发布的包时，在你的模块名字前面加上django-通常很有用。 这有助于其他正在查找Django应用的人区分你的应用是专门用于Django的。
>
> 应用的标签（应用的包的点分路径的最后部分）在INSTALLED_APPS中必须唯一。避免使用与Django的contrib 包 中任何一个使用相同的标签，例如auth、admin和messages。

将polls 目录移动到django-polls目录。

创建一个包含一些内容的文件django-polls/README.rst：

```
django-polls/README.rst
=====
Polls
=====

Polls is a simple Django app to conduct Web-based polls. For eac
h
question, visitors can choose between a fixed number of answers.

Detailed documentation is in the "docs" directory.

Quick start
-----------

1. Add "polls" to your INSTALLED_APPS setting like this::

    INSTALLED_APPS = (
        ...
        'polls',
    )

2. Include the polls URLconf in your project urls.py like this::

    url(r'^polls/', include('polls.urls')),

3. Run `python manage.py migrate` to create the polls models.

4. Start the development server and visit http://127.0.0.1:8000/
admin/
   to create a poll (you'll need the Admin app enabled).

5. Visit http://127.0.0.1:8000/polls/ to participate in the poll
.
```

创建一个django-polls/LICENSE文件。选择License超出本教程的范围，但值得一说的是公开发布的代码如果没有License是毫无用处的。Django和许多与Django兼容的应用以BSD License 发布；然而，你可以随便挑选自己的License。只需要知道你的License的选则将影响谁能够使用你的代码。

下一步我们将创建一个setup.py 文件，它提供如何构建和安装该应用的详细信息。该文件完整的解释超出本教程的范围，setuptools 文档 有很好的解释。创建一个文件django-polls/setup.py，其内容如下：

```
django-polls/setup.py
import os
from setuptools import setup

with open(os.path.join(os.path.dirname(__file__), 'README.rst'))
 as readme:
    README = readme.read()

# allow setup.py to be run from any path
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__)
, os.pardir)))

setup(
    name='django-polls',
    version='0.1',
    packages=['polls'],
    include_package_data=True,
    license='BSD License',  # example license
    description='A simple Django app to conduct Web-based polls.
',
    long_description=README,
    url='http://www.example.com/',
    author='Your Name',
    author_email='yourname@example.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License', # example lice
nse
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        # Replace these appropriately if you are stuck on Python
 2.
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.2',
        'Programming Language :: Python :: 3.3',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)
```

默认只有Python模块和包会包含进包中。如果需要包含额外的文件，我们需要创建一个MANIFEST.in文件。上一步提到的setuptools 文档对这个文件有更详细的讨论。如果要包含模板、README.rst和我们的LICENSE 文件，创建一个文件django-polls/MANIFEST.in，其内容如下：

```
django-polls/MANIFEST.in
include LICENSE
include README.rst
recursive-include polls/static *
recursive-include polls/templates *
```

将详细的文档包含进你的应用中，它是可选的，但建议你这样做。创建一个空的目录django-polls/docs用于将来存放文档。向django-polls/MANIFEST.in添加另外一行：

```
recursive-include docs *
```

注意docs不会包含进你的包中除非你添加一些文件到它下面。许多Django应用还通过类似readthedocs.org这样的站点提供它们的在线文档.

试着通过python setup.py sdist 构建你的包（从django-polls的内部运行）。这创建一个dist目录并构建一个新包django-polls-0.1.tar.gz。

更多关于打包的信息，参见Python 的 打包和分发项目的教程。

使用你自己的包

因为，我们将polls 目录移到项目的目录之外，它不再工作了。我们将通过安装我们的新的django-polls包来修复它。

> 安装成某个用户的库
>
> 以下的步骤将安装django-polls 成某个用户的库。根据用户安装相比系统范围的安装具有许多优点，例如用于没有管理员权限的系统上以及防止你的包影响系统的服务和机器上的其它用户。
>
> 注意根据用户的安装仍然可以影响以该用户身份运行的系统工具，所以virtualenv 是更健壮的解决办法（见下文）。

安装这个包，使用pip（你已经安装好它了，对吧？）：

```
pip install --user django-polls/dist/django-polls-0.1.tar.gz
```

如果幸运，你的Django 项目现在应该可以再次正确工作。请重新运行服务器以证实这点。

若要卸载这个包，使用pip：

```
pip uninstall django-polls
```

发布你的应用：

既然我们已经打包并测试过django-polls，是时候与世界共享它了！要不是它仅仅是个例子，你现在可以：

- 将这个包用邮件发送给朋友。
- 上传这个包到你的网站上。
- 上传这个包到一个公开的仓库，例如Python 包索引 (PyPI)。packaging.python.org has a good tutorial for doing this.

使用 virtualenv 安装Python 包

前面，我们将poll 安装成一个用户的库。它有一些缺点：

- 修改这个用户的库可能影响你的系统上的其它Python 软件。
- 你将不可以运行这个包的多个版本（或者具有相同名字的其它包）。

特别是一旦你维护几个Django项目，这些情况就会出现。如果确实出现，最好的解决办法是使用virtualenv。这个工具允许你维护多个分离的Python环境，每个都具有它自己的库和包的命名空间。

> 译者：Django 文档协作翻译小组，原文：How to write reusable apps。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

为Django编写首个补丁

介绍

有兴趣为社区做出点贡献吗？也许你会在Django中发现你想要修复的漏洞，或者你希望为它添加一个小功能。

为Django作贡献这件事本身就是使你的顾虑得到解决的最好方式。一开始这可能会使你怯步，但事实上是很简单的。整个过程中我们会一步一步为你解说，所以你可以通过例子学习。

本页教程面向的读者

使用教程前，我们希望你至少对于Django的运行方式有基础的了解。这意味着你可以自如地在写你自己的*Django app*时使用教程。 除此之外，你应该对于Python本身有很好的了解。如果您并不太了解， 我们为您推荐Dive Into Python，对于初次使用Python的程序员来说这是一本很棒（而且免费）的在线电子书。

对于版本控制系统及Trac不熟悉的人来说，这份教程及其中的链接所包含的信息足以满足你们开始学习的需求。然而，如果你希望定期为Django贡献，你可能会希望阅读更多关于这些不同工具的信息。

当然对于其中的大部分内容，Django会尽可能做出解释以帮助广大的读者。

何处获得帮助:

如果你在使用本教程时遇到困难，你可以发送信息给*django*开发者 或者登陆 #django-dev on irc.freenode.net 向其他Django使用者需求帮助。

教程包含的内容

一开始我们会帮助你为Django编写补丁，在教程结束时，你将具备对于工具和所包含过程的基本了解。准确来说，我们的教程将包含以下几点：

- 安装Git。
- 如何下载Django的开发副本
- 运行Django的测试组件
- 为你的补丁编写一个测试
- 为你的补丁编码。
- 测试你的补丁。
- 为你所做的改变写一个补丁文件。
- 去哪里寻找更多的信息。

一旦你完成了这份教程，你可以浏览剩下的*Django’s documentation on contributing*. 它包含了大量信息。任何想成为Django的正式贡献者必须去阅读它。如果你有问题，它也许会给你答案

安装Git

使用教程前，你需要安装好Git，下载Django的最新开发版本并且为你作出的改变生成补丁文件

为了确认你是否已经安装了Git, 输入 `git` 进入命令行。如果信息提示命令无法找到, 你就需要下载并安装Git, 详情阅读 Git's download page.

如果你还不熟悉 Git, 你可以在命令行下输入 `git help` 了解更多关于它的命令（确认已安装）。

获取Django 开发版的副本

为Django贡献的第一步就是获取源代码复本。在命令行里， 使用 `cd` 命令进入你想要保存Django的目录

使用下面的命令来下载Django的源码库

```
git clone https://github.com/django/django.git
```

注意

对那些希望使用 virtualenv的人，你可以用:

```
pip install -e /path/to/your/local/clone/django/
```

(你clone的 `django` 目录包含 `setup.py` ），它可以链接到你的cloned确认一个虚拟环境。这是一个伟大的选择，你开发的 Django 副本从您的系统的其余部分隔离，避免了潜在冲突的包。

回滚到更早的Django版本

这个教程中，我们使用 #17549问题来作为学习用例，所以我们要把git中Django的版本回滚到这个问题的补丁没有提交之前。这样的话我们就可以参与到从草稿到补丁的所有过程，包括运行Django的测试套件。

请记住，我们将用**Django**的老版本来到达学习的目的，通常情况下你应当使用当前最新的开发版本来提交补丁。

注意

这个补丁由 Ulrich Petri 开发， Git 提交到 Django 源码 提交id为 ac2052ebc84c45709ab5f0f25e685bf656ce79bc. 因此，我们要回到补丁提交之前的版本号 提交ID： 39f5bc7fc3a4bb43ed8a1358b17fe0521a1a63ac.

首先打开Django源码的根目录（这个目录包含了 `django` , `docs` , `tests` , `AUTHORS` , 等） 然后你你可以根据下面的教程check out老版本的Django：

```
git checkout 39f5bc7fc3a4bb43ed8a1358b17fe0521a1a63ac
```

首先运行Django 的测试套件

当你贡献代码给Django的时候，一个非常重要的问题就是你修改的代码不要给其他部分引入新的bug。 有个办法可以在你更改代码之后检查Django是否能正常工作，就是运行Django的测试套件。如果所有的测试用例都通过，你就有理由相信你的改动完全没有破坏Django。如果你从来没有运行过Django的测试套件，那么比较好的做法是事先运行一遍，熟悉下正常情况下应该输出什么结果。

你可以简单的通过 `cd` 到Django `tests/` 目录下执行测试，如果你是用GNU/Linux, Mac OS X或者其你喜欢的其他Unix系统，执行：

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite
```

如果你使用 Windows，安装 Git 后默认生成 Git 命令行环境："Git Bash" ，在命令行中环境中执行上面的测试命令。GitHub提供了一个很好的教程。

Note

如果你使用了 `virtualenv` ，你可以在执行测试时省略 `PYTHONPATH=..` 。表示在 `测试` .目录的上一层目录寻找 Django 。 `virtualenv` 自动把 Django 放在 `PYTHONPATH` 目录下。

现在坐下来放松一下。Django的整个测试套件有超过4800种不同的测试,所以它运行时间需要5到15分钟,这取决于你的电脑的速度。

Django的测试套件运行时,您将看到一个字符流代表每个测试的运行的状态。 `E` 表示测试中出现异常 和 `F` 表示断言失败。这两种情况都被认为测试失败。同时， `x` 和 `s` 分别表示与期望结果不同和跳过测试。 点表示测试通过。

跳过测试主要由缺少测试所需的外部库引起；查看 *Running all the tests* 获取测所需依赖包，并确保安装由于代码修改造成的新依赖包（这篇教程不需要额外安装依赖包）。

当测试执行完毕后，得到反馈信息显示测试已通过，或者测试失败。因为还没有对 Django 的源码做任何修改，所有的测试用例应该测试通过。如果测试失败或出现错误，回头确认以上执行操作是否正确。查看 *Running the unit tests* 获取更多信息。

注意最新版本 Django 分支不总稳定。当在分支上开发时，你可以查看代码持续集成构建页面的信息 Django's continuous integration builds 来判断测试错误只在你指定的电脑上发生，还是官方版本中也存在该错误。如果点击某个构建信息，可以通过配置列表信息查看错误发生时 Python 以及后端数据库的信息。

Note

在本教程以及所用分支中，测试使用数据库 SQLite 即可， 然而在某些情况下需要测试更多不同的数据库。

为你的ticket写一些测试用例

大多数情况下，Django 的补丁必需包含测试。Bug 修复补丁的测试是一个回归测试，确保该 Bug 不会再次在 Django 中出现。该测试应该在 Bug 存在时测试失败，在 Bug 已经修复后通过测试。新功能补丁的测试必须验证新功能是否正常运行。新功能的测试将在功能正常时通过测试，功能未执行时测试失败。

最好的方式是在修改代码之前写测试单元代码。这种开发风格叫做 测试驱动开发 被应用在项目开发和单一补丁开发过程中。测试单元编写完毕后，执行测试单元，此时测试失败（因为目前还没有修复 BuG 或 添加新功能），如果测试成功通过，你需要重新修改测试单元保证测试失败。然而测试单元并没有阻止 BUG 发生的作用。

现在我们的操作示例。

为分支 #17549 写测试

分支 #17549 描述了以下的额外功能。

> It's useful for URLField to give you a way to open the URL; otherwise you might as well use a CharField.

为了解决这个问题，我们将添加一个 `render` 方法到 `AdminURLFieldWidget` ，通过表单显示一个可点击的链接。在更改代码之前，我们需要一组测试来验证将添加的功能现在以及未来都能正常工作。

进入 Django 下 `tests/regressiontests/admin_widgets/` 目录打开文件 `tests.py` 。在第 269行类 `AdminFileWidgetTest` 之前添加以下内容：

```
class AdminURLWidgetTest(DjangoTestCase):
    def test_render(self):
        w = widgets.AdminURLFieldWidget()
        self.assertHTMLEqual(
            conditional_escape(w.render('test', '')),
            '<input class="vURLField" name="test" type="text" />
'
        )
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example.
com')),
            '<p class="url">Currently:<a href="http://example.co
m">http://example.com</a><br />Change:<input class="vURLField" n
ame="test" type="text" value="http://example.com" /></p>'
        )

    def test_render_idn(self):
        w = widgets.AdminURLFieldWidget()
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example-
äüö.com')),
            '<p class="url">Currently:<a href="http://xn--exampl
e--7za4pnc.com">http://example-äüö.com</a><br />Change:<input cl
ass="vURLField" name="test" type="text" value="http://example-äü
ö.com" /></p>'
        )

    def test_render_quoting(self):
        w = widgets.AdminURLFieldWidget()
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example.
com/<sometag>some text</sometag>')),
            '<p class="url">Currently:<a href="http://example.co
m/%3Csometag%3Esome%20text%3C/sometag%3E">http://example.com/&lt
;sometag&gt;some text&lt;/sometag&gt;</a><br />Change:<input cla
ss="vURLField" name="test" type="text" value="http://example.com
/<sometag>some text</sometag>" /></p>'
        )
        self.assertHTMLEqual(
            conditional_escape(w.render('test', 'http://example-
äüö.com/<sometag>some text</sometag>')),
            '<p class="url">Currently:<a href="http://xn--exampl
e--7za4pnc.com/%3Csometag%3Esome%20text%3C/sometag%3E">http://ex
ample-äüö.com/&lt;sometag&gt;some text&lt;/sometag&gt;</a><br />
Change:<input class="vURLField" name="test" type="text" value="h
ttp://example-äüö.com/<sometag>some text</sometag>" /></p>'
        )
```

该测试会验证我们新添加的方法 `render` 在不同情况下工作正常。

但是这个测试内容看起来比较难...

如果你没有写过测试，第一眼看上去测试代码会有点难。幸运的是测试在编程里是一个 非常 重要的部分， 因此下面有更多的相关信息：

- 为 Django 添加测试代码浏览官网文档 ： 编写和运行测试单元.
- 深入 Python (一个在线免费的 Python 初学者教程) 包含了非常棒的 测试单元介绍.
- 阅读以上文档后，如果想更深入了解测试内容，参考: Python 测试单元文档.

编写新的测试

因为我们还没有对 `AdminURLFieldWidget` 做任何修改，所以我们的测试会失败。我们在 `model_forms_regress` 目录中运行所有测试，确保测试会失败。 在命令行中 `进入` Django 的 `tests/` 目录并执行：

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite admin_wi
dgets
```

如果测试方法运行正常，会出现三个测试失败信息，每个信息对应一个我们刚刚添加的新测试方法。如果所有测试方法都正常通过，请检查上面的测试方法是否添加到了正确的文件位置。

修改 Django 源码

我们在 Django 仓储的标签#17549中添加新功能描述。

为标签#17549编写代码

进入目录 `django/django/contrib/admin/` 并打开文件 `widgets.py` 。 在第302行找到类 `AdminURLFieldWidget` ，在 `__init__` 方法后面添加新方法 `render` 内容如下：

```
def render(self, name, value, attrs=None):
    html = super(AdminURLFieldWidget, self).render(name, value,
attrs)
    if value:
        value = force_text(self._format_value(value))
        final_attrs = {'href': mark_safe(smart_urlquote(value))}
        html = format_html(
            '<p class="url">{} <a {}>{}</a><br />{} {}</p>',
            _('Currently:'), flatatt(final_attrs), value,
            _('Change:'), html
        )
    return html
```

确保测试通过

修改 Django 源码后，我们通过之前编写的测试方法来验证源码修改是否工作正常。运行 `admin_widgets` 目录下所有的测试方法， `进入` Django 的 `tests/` 目录然后运行：

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite admin_wi
dgets
```

哦,好事是我们写了这些测试! 但仍然收到三个测试异常：

```
NameError: global name 'smart_urlquote' is not defined
```

我们忘记导入这些方法。在 `smart_urlquote` 第 13 行的末尾添加 `django/contrib/admin/widgets.py` ，结果如下：

```
from django.utils.html import escape, format_html, format_html_j
oin, smart_urlquote
```

重新运行测试方法正常会通过测试。如果没有通过测试，请重新确认上面提到的类 `AdminURLFieldWidget` 以及新添加的测试方法是否被正确复制到指定位置。

再次运行**Django** 的测试套件

如果已经确认补丁以及测试结果都正常，现在是时候运行 Django 完整的测试用例，验证你的修改是否对 Django 的其他部分造成新的 Bug。 虽然测试用例帮助识别容易被人忽略的错误，但测试通过并不能保证完全没有 Bug 存在。

运行 Django 完整的测试用例， `进入` Django 下 `tests/` 目录并执行：

```
PYTHONPATH=.. python runtests.py --settings=test_sqlite
```

只要没有看到测试异常，你可以继续下一步骤。注意这个修复会产生一个小的 CSS 变动 来格式化新的组件。如果你喜欢，你可以修改它，但是目前我们不做任何修改。

编写文档

这个新功能信息应该被记录到文档。找到文件 `django/docs/ref/models/fields.txt` 第 925 行，在已存在的 `URLField` 文档条目下添加以下内容：

```
.. versionadded:: 1.5

    The current value of the field will be displayed as a clicka
ble link above the
    input widget.
```

关于 `versionadded` 的解释以及文档编写的更多信息，请参考 文档编写 。 这个页面还介绍了怎么在本地重新生成一份文档，你可以查看新生成的 HTML 文档页面.

为你的修改生成补丁

现在是时候生成一个补丁文件，这个补丁文件可以上传到 Trac 或者更新到其他 Django。 运行下面这个命令来查看你的补丁内容：

```
git diff
```

这里显示的内容为当前代码与 check out 时候的代码变化，即之前对代码所做修改前后的变化。

在浏览补丁内容后按 `q` 键退出命令行。如果你的补丁内容看起来正常，运行下面这个命令，在当前目录生成补丁文件：

```
git diff > 17549.diff
```

在 Django 的根目录生成补丁文件 `17549.diff` 。这个补丁文件包含所有的代码变动信息，看起来如下：

```
diff --git a/django/contrib/admin/widgets.py b/django/contrib/ad
min/widgets.py
index 1e0bc2d..9e43a10 100644
--- a/django/contrib/admin/widgets.py
+++ b/django/contrib/admin/widgets.py
@@ -10,7 +10,7 @@ from django.contrib.admin.templatetags.admin_s
tatic import static
 from django.core.urlresolvers import reverse
 from django.forms.widgets import RadioFieldRenderer
 from django.forms.util import flatatt
-from django.utils.html import escape, format_html, format_html_
join
+from django.utils.html import escape, format_html, format_html_
join, smart_urlquote
 from django.utils.text import Truncator
 from django.utils.translation import ugettext as _
 from django.utils.safestring import mark_safe
```

```
@@ -306,6 +306,18 @@ class AdminURLFieldWidget(forms.TextInput):
             final_attrs.update(attrs)
         super(AdminURLFieldWidget, self).__init__(attrs=final_a
ttrs)

+    def render(self, name, value, attrs=None):
+        html = super(AdminURLFieldWidget, self).render(name, va
lue, attrs)
+        if value:
+            value = force_text(self._format_value(value))
+            final_attrs = {'href': mark_safe(smart_urlquote(val
ue))}
+            html = format_html(
+                '<p class="url">{} <a {}>{}</a><br />{} {}</p>'
,
+                _('Currently:'), flatatt(final_attrs), value,
+                _('Change:'), html
+            )
+        return html
+
 class AdminIntegerFieldWidget(forms.TextInput):
     class_name = 'vIntegerField'

diff --git a/docs/ref/models/fields.txt b/docs/ref/models/fields
.txt
index 809d56e..d44f85f 100644
--- a/docs/ref/models/fields.txt
+++ b/docs/ref/models/fields.txt
@@ -922,6 +922,10 @@ Like all :class:`CharField` subclasses, :cl
ass:`URLField` takes the optional
 :attr:`~CharField.max_length`argument. If you don't specify
 :attr:`~CharField.max_length`, a default of 200 is used.

+.. versionadded:: 1.5
+
+The current value of the field will be displayed as a clickable
 link above the
+input widget.

 Relationship fields
 ===================
diff --git a/tests/regressiontests/admin_widgets/tests.py b/test
s/regressiontests/admin_widgets/tests.py
index 4b11543..94acc6d 100644
--- a/tests/regressiontests/admin_widgets/tests.py
+++ b/tests/regressiontests/admin_widgets/tests.py

@@ -265,6 +265,35 @@ class AdminSplitDateTimeWidgetTest(DjangoTe
stCase):
                     '<p class="datetime">Datum: <input value="0
1.12.2007" type="text" class="vDateField" name="test_0" size="10
" /><br />Zeit: <input value="09:30:00" type="text" class="vTime
Field" name="test_1" size="8" /></p>',
```

```
                 )

+class AdminURLWidgetTest(DjangoTestCase):
+    def test_render(self):
+        w = widgets.AdminURLFieldWidget()
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', '')),
+            '<input class="vURLField" name="test" type="text" /
>'
+        )
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example
.com')),
+            '<p class="url">Currently:<a href="http://example.c
om">http://example.com</a><br />Change:<input class="vURLField" 
name="test" type="text" value="http://example.com" /></p>'
+        )
+
+    def test_render_idn(self):
+        w = widgets.AdminURLFieldWidget()
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example
-äüö.com')),
+            '<p class="url">Currently:<a href="http://xn--examp
le--7za4pnc.com">http://example-äüö.com</a><br />Change:<input c
lass="vURLField" name="test" type="text" value="http://example-ä
üö.com" /></p>'
+        )
+
+    def test_render_quoting(self):
+        w = widgets.AdminURLFieldWidget()
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example
.com/<sometag>some text</sometag>')),
+            '<p class="url">Currently:<a href="http://example.c
om/%3Csometag%3Esome%20text%3C/sometag%3E">http://example.com/&l
t;sometag&gt;some text&lt;/sometag&gt;</a><br />Change:<input cl
ass="vURLField" name="test" type="text" value="http://example.co
m/<sometag>some text</sometag>" /></p>'
+        )
+        self.assertHTMLEqual(
+            conditional_escape(w.render('test', 'http://example
-äüö.com/<sometag>some text</sometag>')),
+            '<p class="url">Currently:<a href="http://xn--examp
le--7za4pnc.com/%3Csometag%3Esome%20text%3C/sometag%3E">http://e
xample-äüö.com/&lt;sometag&gt;some text&lt;/sometag&gt;</a><br /
>Change:<input class="vURLField" name="test" type="text" value="
http://example-äüö.com/<sometag>some text</sometag>" /></p>'
+        )

 class AdminFileWidgetTest(DjangoTestCase):
     def test_render(self):
```

接下来做什么？

恭喜，你已经生成了你的第一个 Django 补丁 ！现在你已经明白了整个过程，你可以好好利用这些技能帮助改善Django的代码库。 生成补丁和发送到 Trac 上是有用的，然而我们推荐使用 *面向 git 的工作流*。

目前我们没有在本地对仓储做提交操作，我们可以通过下面这个命令放弃修改并回到最原始 Django 代码状态。

```
git reset --hard HEAD
git checkout master
```

关于新手贡献值的注意事项

在你开始为 Django 编写补丁时，这里有些信息，你应该看一看：

- 你应该阅读了 Django 的参考文档 *claiming tickets and submitting patches*. 它涵盖了Trac 规则，如何声称自己的 tickets，补丁的编码风格和其他一些重要信息。
- 第一次提交补丁额外应该阅读 *documentation for first time contributors*. 这里有很多对新手贡献值的建议。
- 接下来，如果你想对源码贡献有更深入了解，可以阅读接下来的 Django 文档 *Django's documentation on contributing*。 它包含了大量的有用信息，这里可以解决你可能遇到的所有问题。

寻找你的第一个真正的标签

一旦你看过了之前那些信息，你便已经具备了走出困境，为自己编写补丁寻找门票的能力。对于那些有着"容易获得"标准的门票要尤其注意。这些门票实际上常常很简单而且对于第一次撰写补丁的人很有帮助。一旦你熟悉了给Django写补丁，你就可以进一步为更难且更复杂的门票写补丁。

如果你只是想要简单的了解(没人会因此责备你!), 那么你可以尝试着查看这个需要补丁的简单标签列表和已有补丁但需要提升的简单标签列表. 如果你比较擅长写测试，那么你也可以看看这个 需要测试的简单标签列表. 一定要记得遵循在Django的文档声明标签和递交补丁中提到的关于声明标签的指导规则.

接下来要做什么？

一旦一个标签有了补丁，那么它就需要其他人来重审。上传了一个补丁或递交了一个pull request之后，一定记得更新标签的元数据，比如设置标签的标志状态为"has patch"，"doesn't need tests"等。只有这样，其他人才能找到并重审这个标签。从零开始写补丁并不是做贡献的唯一方式。重审一些已经存在的补丁也是一种非常有用的做贡献方式。点击标签鉴别 查看更多详细信息.

模型层

Django 提供了一个抽象层（模型），对您的Web 应用中的数据进行构建及操作。通过以下内容来了解更多:

模型

模型

模型是你的数据的唯一的、权威的信息源。它包含你所储存数据的必要字段和行为。通常，每个模型对应数据库中唯一的一张表。

基础：

- 每个模型都是 `django.db.models.Model` 的一个Python 子类。
- 模型的每个属性都表示数据库中的一个字段。
- Django 提供一套自动生成的用于数据库访问的API；详见执行查询。

简短的例子

这个例子定义一个 `Person` 模型，它有 `first_name` 和 `last_name` 两个属性：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` 和 `last_name` 是模型的两个字段。每个字段都被指定成一个类属性，每个属性映射到一个数据库的列。

上面的 `Person` 模型会在数据库中创建这样一张表：

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

一些技术上的注意事项：

- 这个表的名称 `myapp_person` ，是根据 模型中的元数据自动生成的，也可以覆写为别的名称，详见*Table names*。
- `id` 字段是自动添加的，但这个行为可以被重写。详见自增主键字段。
- 这个例子中的 `CREATE TABLE` SQL 语句使用PostgreSQL 语法格式，要注意的是Django 会根据设置文件 中指定的数据库类型来使用相应的SQL 语句。

使用模型

定义好模型之后，你需要告诉Django 使用这些模型。你要做的就是修改配置文件中的 `INSTALLED_APPS` 设置，在其中添加 `models.py` 所在应用的名称。

例如，如果你的应用的模型位于 `myapp.models` 模块（ `manage.py startapp` 脚本为一个应用创建的包结构）， `INSTALLED_APPS` 部分看上去应该是：

```
INSTALLED_APPS = (
    #...
    'myapp',
    #...
)
```

当你在 `INSTALLED_APPS` 中添加新的应用名时，请确保运行命令 `manage.py migrate` ，可以首先使用 `manage.py makemigrations` 来为它们生成迁移脚本。

字段

模型中不可或缺且最为重要的，就是字段集，它是一组数据库字段的列表。字段被指定为类属性。 要注意选择的字段名称不要和模型 *API* 冲突，比如 `clean` 、 `save` 或者 `delete` 。

例如：

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

字段类型

模型中的每个字段都是 `Field` 子类的某个实例。Django 根据字段类的类型确定以下信息：

- 数据库当中的列类型 (比如， `INTEGER` , `VARCHAR`)。
- 渲染表单时使用的默认HTML 部件（例如， `<input type="text">` , `<select>` ）。
- 最低限度的验证需求，它被用在 Django 管理站点和自动生成的表单中。

Django 自带数十种内置的字段类型；完整字段类型列表可以在模型字段参考 中找到。如果内置类型仍不能满足你的要求，你可以自由地编写符合你要求的字段类型； 详见编写自定义的模型字段。

字段选项

每个字段有一些特有的参数，详见模型字段参考。例如， `CharField` （和它的派生类）需要 `max_length` 参数来指定 `VARCHAR` 数据库字段的大小。

还有一些适用于所有字段的通用参数。 这些参数在参考中有详细定义，这里我们只简单介绍一些最常用的：

`null`

如果为 `True` ，Django 将用 `NULL` 来在数据库中存储空值。 默认值是 `False` .

`blank`

如果为 `True` ，该字段允许不填。默认为 `False` 。

要注意，这与 `null` 不同。 `null` 纯粹是数据库范畴的，而 `blank` 是数据验证范畴的。如果一个字段的 `blank=True` ，表单的验证将允许该字段是空值。如果字段的 `blank=False` ，该字段就是必填的。

`choices`

由二元组组成的一个可迭代对象（例如，列表或元组），用来给字段提供选择项。 如果设置了choices ，默认的表单将是一个选择框而不是标准的文本框，而且这个选择框的选项就是choices 中的选项。

这是一个关于 choices 列表的例子：

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

每个元组中的第一个元素，是存储在数据库中的值；第二个元素是在管理界面或 ModelChoiceField 中用作显示的内容。 在一个给定的 model 类的实例中，想得到某个 choices 字段的显示值，就调用 `get_FOO_display` 方法(这里的 FOO 就是 choices 字段的名称)。例如：

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SI
ZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

`default`

字段的默认值。可以是一个值或者可调用对象。如果可调用 ，每有新对象被创建它都会被调用。

`help_text`

表单部件额外显示的帮助内容。即使字段不在表单中使用，它对生成文档也很有用。

`primary_key`

如果为 `True` ，那么这个字段就是模型的主键。

如果你没有指定任何一个字段的 `primary_key=True` ，Django 就会自动添加一个 `IntegerField` 字段做为主键，所以除非你想覆盖默认的主键行为，否则没必要设置任何一个字段的 `primary_key=True` 。详见自增主键字段。

主键字段是只读的。如果你在一个已存在的对象上面更改主键的值并且保存，一个新的对象将会在原有对象之外创建出来。例如：

```
from django.db import models

class Fruit(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

```
>>> fruit = Fruit.objects.create(name='Apple')
>>> fruit.name = 'Pear'
>>> fruit.save()
>>> Fruit.objects.values_list('name', flat=True)
['Apple', 'Pear']
```

`unique`

如果该值设置为 `True` , 这个数据字段的值在整张表中必须是唯一的

再说一次，这些仅仅是常用字段的简短介绍， 要了解详细内容，请查看 通用 model 字段选项参考(*common model field option reference*).

自增主键字段

默认情况下，Django 会给每个模型添加下面这个字段：

```
id = models.AutoField(primary_key=True)
```

这是一个自增主键字段。

如果你想指定一个自定义主键字段，只要在某个字段上指定 `primary_key=True` 即可。如果 Django 看到你显式地设置了 `Field.primary_key` ，就不会自动添加 `id` 列。

每个模型只能有一个字段指定 `primary_key=True` （无论是显式声明还是自动添加）。

字段的自述名

除 `ForeignKey` 、 `ManyToManyField` 和 `OneToOneField` 之外，每个字段类型都接受一个可选的位置参数 —— 字段的自述名。如果没有给定自述名，Django 将根据字段的属性名称自动创建自述名 —— 将属性名称的下划线替换成空格。

在这个例子中，自述名是 `"person's first name"` :

```
first_name = models.CharField("person's first name", max_length=
30)
```

在这个例子中，自述名是 `"first name"` ：

```
first_name = models.CharField(max_length=30)
```

`ForeignKey`、`ManyToManyField` 和 `OneToOneField` 都要求第一个参数是一个模型类，所以要使用 `verbose_name` 关键字参数才能指定自述名：

```
poll = models.ForeignKey(Poll, verbose_name="the related poll")
sites = models.ManyToManyField(Site, verbose_name="list of sites
")
place = models.OneToOneField(Place, verbose_name="related place"
)
```

习惯上，`verbose_name` 的首字母不用大写。Django 在必要的时候会自动大写首字母。

关系

显然，关系数据库的威力体现在表之间的相互关联。 Django 提供了三种最常见的数据库关系：多对一(many-to-one)，多对多(many-to-many)，一对一(one-to-one)。

多对一关系

Django 使用 `django.db.models.ForeignKey` 定义多对一关系。和使用其它 字段 类型一样：在模型当中把它做为一个类属性包含进来。

`ForeignKey` 需要一个位置参数：与该模型关联的类。

比如，一辆 `Car` 有一个 `Manufacturer` —— 但是一个 `Manufacturer` 生产很多Car，每一辆 `Car` 只能有一个 `Manufacturer` —— 使用下面的定义：

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer)
    # ...
```

你还可以创建递归的关联关系（对象和自己进行多对一关联）和 与尚未定义的模型的关联关系；详见模型字段参考。

建议你用被关联的模型的小写名称做为 `ForeignKey` 字段的名字（例如，上面 `manufacturer` ）。当然，你也可以起别的名字。例如：

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(Manufacturer)
    # ...
```

另见

`ForeignKey` 字段还接受许多别的参数，在模型字段参考有详细介绍。这些选项帮助定义关联关系应该如何工作；它们都是可选的参数。

访问反向关联对象的细节，请见*Following relationships backward example*。

示例代码，请见多对一关系模型示例)。

多对多关系

`ManyToManyField` 用来定义多对多关系，用法和其他 `Field` 字段类型一样：在模型中做为一个类属性包含进来。

`ManyToManyField` 需要一个位置参数：和该模型关联的类。

例如，一个 `Pizza` 可以有多种 `Topping` —— 一种 `Topping` 可以位于多个Pizza 上，而且每个 `Pizza` 可以有多种Topping —— 如下：

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

和使用 `ForeignKey` 一样，你也可以创建递归的关联关系（对象与自己的多对多关联）和与尚未定义关系的模型的关联关系；详见模型字段参考。

建议你以被关联模型名称的复数形式做为 `ManyToManyField` 的名字（例如上例中的 `toppings` ）。

在哪个模型中设置 `ManyToManyField` 并不重要，在两个模型中任选一个即可 —— 不要两个模型都设置。

通常，`ManyToManyField` 实例应该位于可以编辑的表单中。在上面的例子中，`toppings` 位于 `Pizza` 中（而不是在 `Topping` 里面设置 `pizzas` 的 `ManyToManyField` 字段），因为设想一个Pizza 有多种Topping 比一个Topping 位于多个Pizza 上要更加自然。按照上面的方式，在 `Pizza` 的表单中将允许用户选择不同的Toppings。

另见

完整的示例参见多对多关系模型示例。

`ManyToManyField` 字段还接受别的参数，在模型字段参考中有详细介绍。这些选项帮助定义关系应该如何工作；它们都是可选的。

多对多关系中的其他字段

处理类似搭配 pizza 和 topping 这样简单的多对多关系时，使用标准的 `ManyToManyField` 就可以了。但是，有时你可能需要关联数据到两个模型之间的关系上。

例如，有这样一个应用，它记录音乐家所属的音乐小组。我们可以用一个 `ManyToManyField` 表示小组和成员之间的多对多关系。但是，有时你可能想知道更多成员关系的细节，比如成员是何时加入小组的。

对于这些情况，Django 允许你指定一个模型来定义多对多关系。 你可以将其他字段放在中介模型里面。源模型的 `ManyToManyField` 字段将使用 `through` 参数指向中介模型。对于上面的音乐小组的例子，代码如下：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership
')

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

在设置中介模型时，要显式指定外键并关联到多对多关系涉及的模型。这个显式声明定义两个模型之间是如何关联的。

中介模型有一些限制：

- 中介模型必须有且只有一个外键到源模型（上面例子中的 `Group` ），或者你必须使用 `ManyToManyField.through_fields` 显式指定Django 应该使用的

外键。如果你的模型中存在超个一个的外键，并且 `through_fields` 没有指定，将会触发一个无效的错误。 对目标模型的外键有相同的限制（上面例子中的 `Person` ）。
- 对于通过中介模型与自己进行多对多关联的模型，允许存在到同一个模型的两个外键，但它们将被作为多对多关联关系的两个（不同的）方面。如果有超过两个外键，同样你必须像上面一样指定 `through_fields` ，否则将引发一个验证错误。
- 使用中介模型定义与自身的多对多关系时，你必须设置 `symmetrical=False` （详见模型字段参考）。

Changed in Django 1.7:

在Django 1.6 及之前的版本中，中介模型禁止包含多于一个的外键。

既然你已经设置好 `ManyToManyField` 来使用中介模型（在这个例子中就是 `Membership` ），接下来你要开始创建多对多关系。你要做的就是创建中介模型的实例：

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
[<Person: Ringo Starr>]
>>> ringo.group_set.all()
[<Group: The Beatles>]
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
[<Person: Ringo Starr>, <Person: Paul McCartney>]
```

与普通的多对多字段不同，你不能使用 `add` 、 `create` 和赋值语句（比如， `beatles.members = [...]` ）来创建关系：

```
# THIS WILL NOT WORK
>>> beatles.members.add(john)
# NEITHER WILL THIS
>>> beatles.members.create(name="George Harrison")
# AND NEITHER WILL THIS
>>> beatles.members = [john, paul, ringo, george]
```

为什么不能这样做？ 这是因为你不能只创建 `Person` 和 `Group` 之间的关联关系，你还要指定 `Membership` 模型中所需要的所有信息；而简单的 `add` 、 `create` 和赋值语句是做不到这一点的。所以它们不能在使用中介模型

的多对多关系中使用。此时，唯一的办法就是创建中介模型的实例。

`remove()` 方法被禁用也是出于同样的原因。但是 `clear()` 方法却是可用的。它可以清空某个实例所有的多对多关系：

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
[]
```

通过创建中介模型的实例来建立对多对多关系后，你就可以执行查询了。和普通的多对多字段一样，你可以直接使用被关联模型的属性进行查询：

```
# Find all the groups with a member whose name starts with 'Paul
'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

如果你使用了中介模型，你也可以利用中介模型的属性进行查询：

```
# Find all the members of the Beatles that joined after 1 Jan 19
61
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr]
```

如果你需要访问一个成员的信息，你可以直接获取 `Membership` 模型：

```
>>> ringos_membership = Membership.objects.get(group=beatles, pe
rson=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

另一种获取相同信息的方法是，在 `Person` 对象上查询多对多反转关系：

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

一对一关系

`OneToOneField` 用来定义一对一关系。 用法和其他 `字段` 类型一样：在模型里面做为类属性包含进来。

当某个对象想扩展自另一个对象时，最常用的方式就是在这个对象的主键上添加一对一关系。

`OneToOneField` 要一个位置参数：与模型关联的类。

例如，你想建一个“places” 数据库，里面有一些常用的字段，比如address、phone number 等等。 接下来，如果你想在Place 数据库的基础上建立一个Restaurant 数据库，而不想将已有的字段复制到 `Restaurant` 模型，那你可以在 `Restaurant` 添加一个 `OneToOneField` 字段，这个字段指向 `Place` （因为Restaurant 本身就是一个Place；事实上，在处理这个问题的时候，你应该使用一个典型的 继承，它隐含一个一对一关系)。

和使用 `ForeignKey` 一样，你可以定义 递归的关联关系和引用尚未定义关系的模型 。详见模型字段参考。

另见

在一对一关系的模型例子 中有一套完整的例子。

`OneToOneField` 字段也接受一个特定的可选的 `parent_link` 参数，在模型字段参考 中有详细介绍。

在以前的版本中， `OneToOneField` 字段会自动变成模型 的主键。不过现在已经不这么做了(不过要是你愿意的话，你仍可以传递 `primary_key` 参数来创建主键字段)。所以一个 模型 中可以有多个 `OneToOneField` 字段。

跨文件的模型

访问其他应用的模型是非常容易的。 在文件顶部你定义模型的地方，导入相关的模型来实现它。然后，无论在哪里需要的话，都可以引用它。例如：

```
from django.db import models
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(ZipCode)
```

字段命名的限制

Django 对字段的命名只有两个限制：

1. 字段的名称不能是Python 保留的关键字，因为这将导致一个Python 语法错误。例如：

```
class Example(models.Model):
    pass = models.IntegerField() # 'pass' is a reserved word
!
```

2. 由于Django 查询语法的工作方式，字段名称中连续的下划线不能超过一个。例如：

```
class Example(models.Model):
    foo__bar = models.IntegerField() # 'foo__bar' has two un
derscores!
```

这些限制有变通的方法，因为没有要求字段名称必须与数据库的列名匹配。参 `db_column` 选项。

SQL 的保留字例如 `join` 、 `where` 和 `select` ，可以用作模型的字段名，因为 Django 会对底层的SQL 查询语句中的数据库表名和列名进行转义。 它根据你的数据库引擎使用不同的引用语法。

自定义字段类型

如果已有的模型字段都不合适，或者你想用到一些很少见的数据库列类型的优点，你可以创建你自己的字段类型。创建你自己的字段在编写自定义的模型字段中有完整讲述。

元选项

使用内部的 `class Meta` 定义模型的元数据，例如：

```
from django.db import models

class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

模型元数据是“任何不是字段的数据”，比如排序选项（ `ordering` ），数据表名（ `db_table` ）或者人类可读的单复数名称（ `verbose_name` 和 `verbose_name_plural` ）。在模型中添加 `class Meta` 是完全可选的，所有选项都不是必须的。

所有 `元` 选项的完整列表可以在模型选项参考找到。

模型的属性

`objects`

The most important attribute of a model is the `Manager`. It's the interface through which database query operations are provided to Django models and is used to *retrieve the instances* from the database. If no custom `Manager` is defined, the default name is `objects`. Managers are only accessible via model classes, not the model instances.

模型的方法

可以在模型上定义自定义的方法来给你的对象添加自定义的"底层"功能。`Manager` 方法用于"表范围"的事务，模型的方法应该着眼于特定的模型实例。

这是一个非常有价值的技术，让业务逻辑位于同一个地方 —— 模型中。

例如，下面的模型具有一些自定义的方法：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

这个例子中的最后一个方法是一个*property*。

模型实例参考 具有一个完整的为模型自动生成的方法 列表。你可以覆盖它们 —— 参见下文覆盖模型预定义的方法 —— 但是有些方法你会始终想要重新定义：

`__str__()` (Python 3)

Python 3 equivalent of `__unicode__()` .

`__unicode__()` (Python 2)

一个Python "魔法方法"，返回对象的Unicode "表示形式"。当模型实例需要强制转换并显示为普通的字符串时，Python 和Django 将使用这个方法。最明显是在交互式控制台或者管理站点显示一个对象的时候。

将将永远想要定义这个方法；默认的方法几乎没有意义。

`get_absolute_url()`

它告诉Django 如何计算一个对象的URL。Django 在它的管理站点中使用到这个方法，在其它任何需要计算一个对象的URL 时也将用到。

任何具有唯一标识自己的URL 的对象都应该定义这个方法。

覆盖预定义的模型方法

还有另外一部分封装数据库行为的模型方法，你可能想要自定义它们。特别是，你将要经常改变 `save()` 和 `delete()` 的工作方式。

你可以自由覆盖这些方法（和其它任何模型方法）来改变它们的行为。

覆盖内建模型方法的一个典型的使用场景是，你想在保存一个对象时做一些其它事情。例如（参见 `save()` 中关于它接受的参数的文档）：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real
" save() method.
        do_something_else()
```

你还可以阻止保存：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super(Blog, self).save(*args, **kwargs) # Call the "
real" save() method.
```

必须要记住调用超类的方法—— `super(Blog, self).save(*args, **kwargs)` —— 来确保对象被保存到数据库中。如果你忘记调用超类的这个方法，默认的行为将不会发生且数据库不会有任何改变。

还要记住传递参数给这个模型方法 —— 即 `*args, **kwargs` 。 Django 未来将一直会扩展内建模型方法的功能并添加新的参数。如果在你的方法定义中使用 `*args, **kwargs` ，将保证你的代码自动支持这些新的参数。

Overridden model methods are not called on bulk operations

注意，当使用查询集批量删除对象时，将不会为每个对象调用 `delete()` 方法。为确保自定义的删除逻辑得到执行，你可以使用 `pre_delete` 和/或 `post_delete` 信号。

不幸的是，当批量 `creating` 或 `updating` 对象时没有变通方法，因为不会调用 `save()` 、 `pre_save` 和 `post_save` 。

执行自定义的SQL

另外一个常见的需求是在模型方法和模块级别的方法中编写自定义的SQL 语句。关于使用原始SQL 语句的更多细节，参见使用原始 *SQL* 的文档。

模型继承

Django 中的模型继承与 Python 中普通类继承方式几乎完全相同，但是本页头部列出的模型基本的要求还是要遵守。这表示自定义的模型类应该继承 `django.db.models.Model` 。

你唯一需要作出的决定就是你是想让父模型具有它们自己的数据库表，还是让父模型只持有一些共同的信息而这些信息只有在子模型中才能看到。

在Django 中有3中风格的继承。

1. 通常，你只想使用父类来持有一些信息，你不想在每个子模型中都敲一遍。这个类永远不会单独使用，所以你使用抽象基类。

2. 如果你继承一个已经存在的模型且想让每个模型具有它自己的数据库表，那么应该使用多表继承。
3. 最后，如果你只是想改变模块Python 级别的行为，而不用修改模型的字段，你可以使用代理模型。

抽象基类

当你想将一些常见信息存储到很多model的时候，抽象化类是十分有用的。你编写完基类之后，在 *Meta*类中设置 `abstract=True` ，该类就不能创建任何数据表。取而代之的是，当它被用来作为一个其他model的基础类时，它将被加入那一子类中。如果抽象化基础类和它的子类有相同的项，那么将会出现error（并且Django将返回一个exception）。

一个例子

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

`Student` 模型将有三个项： `name` , `age` 和 `home_group` 。 `CommonInfo` 模型无法像一般的Django模型一样使用，因为它是一个抽象化基础类。它无法生成数据表单或者管理器，并且不能实例化或者储存。

对很多用户来说, 这种类型的模型继承就是你想要的。它提供一种在 Python 语言层级上提取公共信息的方式，但在数据库层级上，各个子类仍然只创建一个数据库。

`元` 继承

当一个抽象类被创建的时候, Django会自动把你在基类中定义的 *Meta* 作为子类的一个属性。如果子类没有声明自己的*Meta* 类, 他将会继承父类的*Meta*. 如果子类想要扩展父类 的，可以继承父类的 *Meta* 即可，例如

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

继承时，Django 会对基类的 *Meta*类做一个调整：在安装 *Meta*属性之前，Django 会设置 `abstract=False` 。这意味着抽象基类的子类不会自动变成抽象类。 当然，你可以让一个抽象类继承另一个抽象基类，不过每次都要显式地设置 `abstract=True` 。

对于抽象基类而言，有些属性放在 *Meta* 内嵌类里面是没有意义的。例如，包含 `db_table` 将意味着所有的子类(是指那些没有指定自己的 *Meta* 类的子类)都使用同一张数据表，一般来说，这并不是我们想要的。

小心使用 `related_name`

如果你在 `ForeignKey` 或 `ManyToManyField` 字段上使用 `related_name` 属性，你必须总是为该字段指定一个唯一的反向名称。但在抽象基类上这样做就会引发一个很严重的问题。因为 Django 会将基类字段添加到每个子类当中，而每个子类的字段属性值都完全相同 (这里面就包括 `related_name`)。

当你在(且仅在)抽象基类中使用 `related_name` 时，如果想绕过这个问题，名称中就要包含 `'%(app_label)s'` 和 `'%(class)s'` 。

- `'%(class)s'` 会替换为子类的小写加下划线格式的名称，字段在子类中使用。
- `'%(app_label)s'` 会替换为应用的小写加下划线格式的名称，应用包含子类。每个安装的应用名称都应该是唯一的，而且应用里每个模型类的名称也应该是唯一的，所以产生的名称应该彼此不同。

例如，假设有一个app叫做 `common/models.py` ：

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(OtherModel, related_name="%(app
_label)s_%(class)s_related")

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

以及另一个应用 `rare/models.py` ：

```
from common.models import Base

class ChildB(Base):
    pass
```

ChildA.m2m 字段的反向名称是 childa_related，而 ChildB.m2m 字段的反向名称是 childb_related。这取决于你如何使用 `'%(class)s'` 和 `'%(app_label)s` 来构造你的反向名称。如果你没有这样做，Django 就会在验证 model (或运行 `migrate`) 时抛出错误。

果你没有在抽象基类中为某个关联字段定义 `related_name` 属性，那么默认的反向名称就是子类名称加上 `'_set'` ，它能否正常工作取决于你是否在子类中定义了同名字段。例如，在上面的代码中，如果去掉 `related_name` 属性，在 `ChildA` 中， `m2m` 字段的反向名称就是 `childa_set` ；而 `ChildB` 的 m2m 字段的反向名称就是 `childb_set` 。

多表继承

这是 Django 支持的第二种继承方式。使用这种继承方式时，同一层级下的每个子 model 都是一个真正意义上完整的 model 。 每个子 model 都有专属的数据表，都可以查询和创建数据表。 继承关系在子 model 和它的每个父类之间都添加一个链接 (通过一个自动创建的 `OneToOneField` 来实现)。 例如：

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

`Place` 里面的所有字段在 `Restaurant` 中也是有效的，只不过数据保存在另外一张数据表当中。所以下面两个语句都是可以运行的：

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

如果你有一个 `Place` ，那么它同时也是一个 `Restaurant` ，那么你可以使用子 model 的小写形式从 Pl `Place` 对象中获得与其对应的 `Restaurant` 对象：

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

但是，如果上例中的 `p` 并不是 `Restaurant` (比如它仅仅只是 `Place` 对象，或者它是其他类的父类)，那么在引用 `p.restaurant` 就会抛出 `Restaurant.DoesNotExist` 异常。

多表继承中的 `Meta`

在多表继承中，子类继承父类的 *Meta*类是没什么意义的。所有的 *Meta* 选项已经对父类起了作用，再次使用只会起反作用。(这与使用抽象基类的情况正好相反，因为抽象基类并没有属于它自己的内容)

所以子 model 并不能访问它父类的 *Meta* 类。但是在某些受限的情况下，子类可以从父类继承某些 Meta ：如果子类没有指定 `ordering` 属性或 `get_latest_by` 属性，它就会从父类中继承这些属性。

如果父类有了排序设置，而你并不想让子类有任何排序设置，你就可以显式地禁用排序：

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

继承与反向关联

因为多表继承使用了一个隐含的 `OneToOneField` 来链接子类与父类，所以象上例那样，你可以用父类来指代子类。但是这个 OnetoOneField 字段默认的 `related_name` 值与 `ForeignKey` 和 `ManyToManyField` 默认的反向名称相同。如果你与其他 model 的子类做多对一或是多对多关系，你就必须在每个多对一和多对多字段上强制指定 `related_name` 。如果你没这么做，Django 就会在你运行 验证(validation) 时抛出异常。

例如，仍以上面 `Place` 类为例，我们创建一个带有 `ManyToManyField` 字段的子类：

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

这会产生一个错误：

```
Reverse query name for 'Supplier.customers' clashes with reverse
 query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition fo
r
'Supplier.customers' or 'Supplier.place_ptr'.
```

像下面那样，向 `customers` 字段中添加 `related_name` 可以解决这个错误： `models.ManyToManyField(Place, related_name='provider')` 。

指定链接父类的字段

之前我们提到，Django 会自动创建一个 `OneToOneField` 字段将子类链接至非抽象的父 model 。如果你想指定链接父类的属性名称，你可以创建你自己的 `OneToOneField` 字段并设置 `parent_link=True` ，从而使用该字段链接父类。

代理模型

使用 多表继承时，model 的每个子类都会创建一张新数据表，通常情况下，这正是我们想要的操作。这是因为子类需要一个空间来存储不包含在基类中的字段数据。 但有时，你可能只想更改 model 在 Python 层的行为实现。比如：更改默认的 manager ，或是添加一个新方法。

而这，正是代理 model 继承方式要做的：为原始 model 创建一个代理 。你可以创建，删除，更新代理 model 的实例，而且所有的数据都可以象使用原始 model 一样被保存。 不同之处在于：你可以在代理 model 中改变默认的排序设置和默认的 manager ，更不会对原始 model 产生影响。

声明代理 model 和声明普通 model 没有什么不同。 设置 `Meta` 类中 `proxy` 的值为 `True` ，就完成了对代理 model 的声明。

举个例子，假设你想给 Django 自带的标准 `Person` model添加一个方法。你可以这样做：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

`MyPerson` 类和它的父类 `Person` 操作同一个数据表。特别的是， `Person` 的任何实例也可以通过 `MyPerson` 访问，反之亦然：

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

你也可以使用代理 model 给 model 定义不同的默认排序设置。 你可能并不想每次都给 `Person` 模型排序，但是使用代理的时候总是按照 `last_name` 属性排序。这非常容易：

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

现在，普通的 `Person` 查询时无序的，而 `OrderedPerson` 查询会按照 `last_name` 排序。

查询集始终返回请求的模型

也就是说，没有办法让DJango在查询 `Person` 对象时返回 `MyPerson` 对象。 `Person` 对象的查询集会返回相同类型的对象。代理对象的要点是，依赖于原生 `Person` 对象的代码仍然使用它，而你可以使用你添加进来的扩展对象（它不会依赖其它任何代码）。而并不是将 `Person` 模型（或者其它）在所有地方替换为其它你自己创建的模型。

基类的限制

代理 模型必须继承自一个非抽象基类。 你不能继承自多个非抽象基类，这是因为一个代理 model 不能连接不同的数据表。 代理 model 也可以继承任意多个抽象基类，但前提是它们没有 定义任何 model 字段。

代理模型的管理器

如果你没有在代理 模型中定义任何 管理器 ，代理模型就会从父类中继承 管理器 。如果你在代理 模型中定义了一个 管理器 ，它就会变成默认的管理器 ，不过定义在父类中的管理器仍然有效。

继续上面的例子，当你查询 `Person` 模型的时候，你可以改变默认 管理器，例如：

```
from django.db import models

class NewManager(models.Manager):
    # ...
    pass

class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

如果你想要向代理中添加新的管理器，而不是替换现有的默认管理器，你可以使用自定义管理器管理器文档中描述的技巧：创建一个含有新的管理器的基类，并且在主基类之后继承它：

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

你可能不需要经常这样做，但这样做是可行的。

代理 模型与非托管 模型之间的差异

代理 model 继承看上去和使用 `Meta` 类中的 `managed` 属性的非托管 model 非常相似。但两者并不相同，你应当考虑选用哪种方案。

一个不同之处是你可以在 `Meta.managed=False` 的 model 中定义字段(事实上，是必须指定，除非你真的想得到一个空 model)。在创建非托管 model 时要谨慎设置 `Meta.db_table` ，这是因为创建的非托管 model 映射某个已存在的 model ，并且有自己的方法。因此，如果你要保证这两个 model 同步并对程序进行改动，那么就会变得繁冗而脆弱。

另一个不同之处是两者对 管理器的处理方式不同。 代理 model 要与它所代理的 model 行为相似，所以代理 model 要继承父 model 的 managers ，包括它的默认 manager 。 但在普通的多表继承中，子类不能继承父类的 manager ，这是因为在处理非基类字段时，父类的 manager 未必适用。 后一种情况在 管理器文档有详细介绍。

我们实现了这两种特性之后，曾尝试把两者结合到一起。 结果证明，宏观的继承关系和微观的 管理器揉在一起，不仅导致 API 复杂难用，而且还难以理解。 由于任何场合下都可能需要这两个选项，所以目前二者仍是各自独立使用的。

所以，一般规则是：

1. 如果你要借鉴一个已有的 模型或数据表，且不想涉及所有的原始数据表的列，那就令 `Meta.managed=False` 。通常情况下，对数据库视图创建 模型或是数据表不需要由 Django 控制时，就使用这个选项。
2. 如果你想对 model 做 Python 层级的改动，又想保留字段不变，那就令 `Meta.proxy=True` 。因此在数据保存时，代理 model 相当于完全复制了原始模型的存储结构。

多重继承

就像Python的子类那样，DJango的模型可以继承自多个父类模型。切记一般的Python名称解析规则也会适用。出现特定名称的第一个基类(比如*Meta*)是所使用的那个。例如，这意味着如果多个父类含有 *Meta*类，只有第一个会被使用，剩下的

会忽略掉。

一般来说，你并不需要继承多个父类。多重继承主要对“mix-in”类有用：向每个继承mix-in的类添加一个特定的、额外的字段或者方法。你应该尝试将你的继承关系保持得尽可能简洁和直接，这样你就不必费很大力气来弄清楚某段特定的信息来自哪里。

Changed in Django 1.7.

Django 1.7之前，继承多个含有 `id` 主键字段的模型不会抛出异常，但是会导致数据丢失。例如，考虑这些模型（由于 `id` 字段的冲突，它们不再有效）：

```
class Article(models.Model):
    headline = models.CharField(max_length=50)
    body = models.TextField()

class Book(models.Model):
    title = models.CharField(max_length=50)

class BookReview(Book, Article):
    pass
```

这段代码展示了如何创建子类的对象，并覆写之前创建的父类对象中的值。

```
>>> article = Article.objects.create(headline='Some piece of new
s.')
>>> review = BookReview.objects.create(
...     headline='Review of Little Red Riding Hood.',
...     title='Little Red Riding Hood')
>>>
>>> assert Article.objects.get(pk=article.pk).headline == articl
e.headline
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AssertionError
>>> # the "Some piece of news." headline has been overwritten.
>>> Article.objects.get(pk=article.pk).headline
'Review of Little Red Riding Hood.'
```

你可以在模型基类中使用显式的 `AutoField` 来合理使用多重继承：

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...

class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...

class BookReview(Book, Article):
    pass
```

或者是使用一个公共的祖先来持有 `AutoField` ：

```
class Piece(models.Model):
    pass

class Article(Piece):
    ...

class Book(Piece):
    ...

class BookReview(Book, Article):
    pass
```

Field name “hiding” is not permitted

普通的 Python 类继承允许子类覆盖父类的任何属性。 但在 Django 中，重写 `Field` 实例是不允许的(至少现在还不行)。如果基类中有一个 `author` 字段，你就不能在子类中创建任何名为 `author` 的字段。

重写父类的字段会导致很多麻烦，比如：初始化实例(指定在 `Model.__init__` 中被实例化的字段) 和序列化。而普通的 Python 类继承机制并不能处理好这些特性。所以 Django 的继承机制被设计成与 Python 有所不同，这样做并不是随意而为的。

这些限制仅仅针对做为属性使用的 `Field` 实例，并不是针对 Python 属性，Python 属性仍是可以被重写的。 在 Python 看来，上面的限制仅仅针对字段实例的名称：如果你手动指定了数据库的列名称，那么在多重继承中，你就可以在子类和某个祖先类当中使用同一个列名称。(因为它们使用的是两个不同数据表的字段)。

如果你在任何一个祖先类中重写了某个 model 字段，Django 都会抛出 `FieldError` 异常。

另见

The Models Reference

Covers all the model related APIs including model fields, related objects, and `QuerySet`.

> 译者：Django 文档协作翻译小组，原文：Model syntax。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

模型字段参考

本文档包含了Django提供的全部模型 `字段` 的字段选项 和 字段类型的API参考。

请看:

如果内建的字段不能满足你的需要，你可以尝试包含对特定国家和文化有帮助的配套代码的 django-localflavor。当然，你也可以很容易的编写你自定义的字段。

注意

严格意义上来讲，Model 是定义在 `django.db.models.fields` 里面，但为了使用方便，它们被导入到 `django.db.models` 中；标准上，我们导入 `from django.db import models` ，然后使用? `models.<Foo>Field` 的形式使用字段。

字段选项(Field options)

下列参数是全部字段类型都可用的，而且都是可选择的。

null

`Field.null`

如果为 `True` ，Django 将空值以 `NULL` 存储到数据库中。默认值是 `False` 。

字符串字段例如 `CharField` 和 `TextField` 要避免使用 `null` ，因为空字符串值将始终储存为空字符串而不是 `NULL` 。如果字符串字段的 `null=True` ，那意味着对于“无数据”有两个可能的值： `NULL` 和空字符串。在大多数情况下，对于“无数据”声明两个值是赘余的，Django 的惯例是使用空字符串而不是 `NULL` 。

无论是字符串字段还是非字符串字段，如果你希望在表单中允许空值，你将还需要设置 `blank=True` ，因为 `null` 仅仅影响数据库存储（参见 `blank` ）。

注意

在使用Oracle 数据库时，数据库将存储 `NULL` 来表示空字符串，而与这个属性无关。

如果你希望 `BooleanField` 接受 `null` 值，请用 `NullBooleanField` 代替。

blank

`Field.blank`

如果为 `True` ，则该字段允许为空白。 默认值是 `False` 。

注意它与 `null` 不同。 `null` 纯粹是数据库范畴的概念，而 `blank` 是数据验证范畴的。如果字段设置 `blank=True` ，表单验证时将允许输入空值。如果字段设置 `blank=False` ，则该字段为必填。

choices

`Field.choices`

它是一个可迭代的结构(比如，列表或是元组)，由可迭代的二元组组成(比如 `[(A, B), (A, B) ...]`)，用来给这个字段提供选择项。如果设置了 choices ，默认表格样式就会显示选择框，而不是标准的文本框，而且这个选择框的选项就是 choices 中的元组。

每个元组中的第一个元素，是存储在数据库中的值；第二个元素是该选项更易理解的描述。 比如:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
)
```

一般来说，最好在模型类内部定义choices，然后再给每个值定义一个合适名字的常量。

```
from django.db import models

class Student(models.Model):
    FRESHMAN = 'FR'
    SOPHOMORE = 'SO'
    JUNIOR = 'JR'
    SENIOR = 'SR'
    YEAR_IN_SCHOOL_CHOICES = (
        (FRESHMAN, 'Freshman'),
        (SOPHOMORE, 'Sophomore'),
        (JUNIOR, 'Junior'),
        (SENIOR, 'Senior'),
    )
    year_in_school = models.CharField(max_length=2,
                                      choices=YEAR_IN_SCHOOL_CHO
ICES,
                                      default=FRESHMAN)

    def is_upperclass(self):
        return self.year_in_school in (self.JUNIOR, self.SENIOR)
```

尽管你可以在模型类的外部定义choices然后引用它，但是在模型类中定义choices和其每个choice的name(即元组的第二个元素)可以保存所有使用choices的类的信息， 也使得choices更容易被应用（例如， `Student.SOPHOMORE` 可以在任何引入 `Student` 模型的位置生效)。

你也可以归类可选的choices到已命名的组中用来达成组织整理的目的:

```
MEDIA_CHOICES = (
    ('Audio', (
            ('vinyl', 'Vinyl'),
            ('cd', 'CD'),
        )
    ),
    ('Video', (
            ('vhs', 'VHS Tape'),
            ('dvd', 'DVD'),
        )
    ),
    ('unknown', 'Unknown'),
)
```

每个元组的第一个元素是组的名字。第二个元素是一组可迭代的二元元组，每一个二元元组包含一个值和一个给人看的名字构成一个选项。分组的选项可能会和未分组的选项合在同一个list中。（就像例中的*unknown*选项）。

对于有 `choices` set的模型字段, Django 将会加入一个方法来获取当前字段值的易于理解的名称(即元组的第二个值)参见数据库API文档中的 `get_FOO_display()` 。

请注意choices可以是任何可迭代的对象 – 不是必须是列表或者元组。这一点使你可以动态的构建choices。但是如果你发现你自己搞不定动态的 `choices` ，你最好还是使用 `ForeignKey` 来构建一个合适的数据库表。如果有数据变动的话， `choices` 意味着那些变动不多的静态数据。

New in Django 1.7.

除非 `blank=False` 和 `default` 一起在字段中被设置，否则，可选择菜单将会有 `"---------"` 的标签。要重写这个行为, 需要加入一个包含 `None` 的元组到 `choices` 里面; 例如 `(None, 'Your String For Display')` . 或者, 你可以在操作有意义的地方用一个空字符串代替 `None` ?- 比如在一个 `CharField` .

db_column

`Field.db_column`

数据库中用来表示该字段的名称。如果未指定，那么Django将会使用Field名作为字段名.

如果你的数据库列名为SQL语句的保留字，或者是包含不能作为Python 变量名的字符，如连字符，没问题。Django 会在后台给列名和表名加上双引号。

db_index

`Field.db_index`

若值为 `True`，则 `django-admin sqlindexes` 将会为此字段输出 `CREATE INDEX` 语句。（译注：为此字段创建索引）

db_tablespace

`Field.db_tablespace`

?如果该字段有索引的话，*database tablespace*的名称将作为该字段的索引名。 如果 `DEFAULT_INDEX_TABLESPACE` 已经设置，则默认值是由DEFAULT_INDEX_TABLESPACE指定, 如果没有设置则由 `db_tablespace` 指定，如果后台数据库不支持表空间，或者索引，则该选项被忽略

default

`Field.default`

该字段的默认值. 它可以是一个值或者一个可调用对象. 如果是一个可调用对象，那么在每一次创建新对象的时候，它将会调用一次.

这个默认值不可以是一个可变对象（如字典，列表，等等）,因为对于所有模型的一个新的实例来说，它们指向同一个引用。或者，把他们包装为一个可调用的对象。例如，你有一个自定义的 `JSONField`，并且想指定一个特定的字典值，可以如下使用：

```
def contact_default():
    return {"email": "to1@example.com"}

contact_info = JSONField("ContactInfo", default=contact_default)
```

请注意 `lambda` s 函数不可作为如 `default` 这类可选参数的值.因为它们无法被 *migrations命令序列化*. 请参见文档其他部分。

默认值会在新实例创建并且没有给该字段提供值时使用。如果字段为主键，默认值也会在设置为 `None` 时使用。

Changed in Django 1.8:

之前的版本不会使用 `None` 作为主键?

editable

`Field.editable`

如果设为 `False` , 这个字段将不会出现在 admin 或者其他 `ModelForm`. 他们也会跳过 模型验证. 默认是 `True` .

error_messages

`Field.error_messages`

`error_messages` 参数能够让你重写默认抛出的错误信息。通过指定 key 来确认你要重写的错误信息。

error_messages 的 key 值包括 `null` , `blank` , `invalid` , `invalid_choice` , `unique` , 和 `unique_for_date` . 其余的 error_messages 的 keys 是不一样的在不同的章节下 Field types 。

New in Django 1.7.

这个 `unique_for_date` 的 error_messages 的key 是在 1.7 中加的。

help_text

`Field.help_text`

额外的 'help' 文本将被显示在表单控件form中。即便你的字段没有应用到一个form里面，这样的操作对文档化也很有帮助。

注意这 不 会自动添加 HTML 标签。需要你在 `help_text` 包含自己需要的格式。例如:

```
help_text="Please use the following format: <em>YYYY-MM-DD</em>.
"
```

另外, 你可以使用简单文本和 `django.utils.html.escape()` 以避免任何HTML特定的字符.请确保你所使用的help text能够避免那些由不受信任的用户进行的跨站点脚本攻击。

primary_key

`Field.primary_key`

若为 `True` , 则该字段会成为模型的主键字段。

如果你没有在模型的任何字段上指定 `primary_key=True` , Django会自动添加一个 `AutoField` 字段来充当主键。 所以除非你想要覆盖默认的主键行为，否则不需要在任何字段上设定 `primary_key=True` 。更多内容 请参考 *Automatic primary key fields*.

`primary_key=True` 暗含着 `null=False` 和 `unique=True`. 一个对象上只能拥有一个主键.

主键字段是只读的。如果你改变了一个已存在对象上的主键并且保存的话，会创建一个新的对象，而不是覆盖旧的.

unique

`Field.unique`

如果为? `True`, 这个字段在表中必须有唯一值.

这是一个在数据库级别的强制性动作,并且通过模型来验证。如果你试图用一个重复的值来保存 `unique` 字段，那么一个 `django.db.IntegrityError` 就会通过模型的 `save()` 函数抛出来。

除了 `ManyToManyField`、`OneToOneField` 和 `FileField` 以外的其他字段类型都可以使用这个设置。

注意当设置 `unique` 为 `True` 时，你不需要再指定 `db_index`，因为 `unique` 本身就意味着一个索引的创建。

unique_for_date

`Field.unique_for_date`

当设置它为 `DateField` 和 `DateTimeField` 字段的名称时，表示要求该字段对于相应的日期字段值是唯一的。

例如，你有一个 `title` 字段设置 `unique_for_date="pub_date"`，那么Django将不允许两个记录具有相同的 `title` 和 `pub_date`。

注意，如果你将它设置为 `DateTimeField`，只会考虑其日期部分。此外，如果 `USE_TZ` 为 `True`，检查将以对象保存时的当前的时区 进行。

这是在模型验证期间通过 `Model.validate_unique()` 强制执行的，而不是在数据库层级进行验证。如果 `unique_for_date` 约束涉及的字段不是 `ModelForm` 中的字段（例如，`exclude` 中列出的字段或者设置了 `editable=False`），`Model.validate_unique()` 将忽略该特殊的约束。

unique_for_month

`Field.unique_for_month`

类似 `unique_for_date`，只是要求字段对于月份是唯一的。

unique_for_year

`Field.unique_for_year`

类似 `unique_for_date` 和 `unique_for_month` 。

verbose_name

`Field.verbose_name`

一个字段的可读性更高的名称。如果用户没有设定冗余名称字段，Django会自动将该字段属性名中的下划线转换为空格，并用它来创建冗余名称。可以参照 *Verbose field names*.

validators

`Field.validators`

该字段将要运行的一个Validator 的列表。更多信息参见*Validators 的文档* 。

注册和获取查询

`Field` 实现了*查询注册API*。该API 可以用于自定义一个字段类型的可用的查询，以及如何从一个字段获取查询。

字段类型 （Field types）

自增字段

class `AutoField` (***options*)

一个根据实际ID自动增长的 `IntegerField` . 你通常不需要直接使用;如果不指定,一个主键字段将自动添加到你创建的模型中.详细查看?主键字段.

BigIntegerField

class `BigIntegerField` ([***options*])

一个64位整数, 类似于一个? `IntegerField` ,它的值的范围是? `-9223372036854775808` 到 `9223372036854775807` 之间. 这个字段默认的表单组件是一个 `TextInput` .

BinaryField

class `BinaryField` ([***options*])

这是一个用来存储原始二进制码的Field. 只支持 `bytes` 赋值，注意这个Field只有很有限的功能。例如，不大可能在一个 `BinaryField` 值的数据上进行查询

Abusing `BinaryField`

尽管你可能想使用数据库来存储你的文件，但是99%的情况下这都是不好的设计。这个字段 不是替代*static files* 的合理操作.

BooleanField

class `BooleanField` (***options*)

true/false 字段。

此字段的默认表单挂件是一个 `CheckboxInput` .

如果你需要设置 `null` 值，则使用 `NullBooleanField` 来代替BooleanField。

如果 `Field.default` 没有指定的话， `BooleanField` 的默认值是 `None` 。

CharField

class `CharField` (*max_length=None*[, ***options*])

一个用来存储从小到很大各种长度的字符串的地方

如果是巨大的文本类型, 可以用 `TextField` .

这个字段默认的表单样式是 `TextInput` .

`CharField` 必须接收一个额外的参数:

`CharField.max_length`

字段的最大字符长度.max_length将在数据库层和Django表单验证中起作用, 用来限定字段的长度.?

Note

如果你在写一个需要导出到多个不同数据库后端的应用，你需要注意某些后端对 `max_length` 有一些限制，查看数据库后端注意事项来获取更多的细节

MySQL的使用者们:

如果你在使用该字段的同时使用MySQLdb 1.2.2以及 `utf8_bin` 校对 其一般不是默认情况), 你必须了解一些问题. 详细请参考 *MySQL database notes* .

CommaSeparatedIntegerField

class `CommaSeparatedIntegerField` (*max_length=None*[, ***options*])

一个逗号分隔的整数字段。像 `CharField` 一样， 需要一个 `max_length` 参数，同时数据库移植时也需要注意。

DateField

class `DateField` ([*auto_now=False*, *auto_now_add=False*, ***options*])

这是一个使用Python的 `datetime.date` 实例表示的日期. 有几个额外的设置参数:

`DateField.auto_now`

每次保存对象时，自动设置该字段为当前时间。用于"最后一次修改"的时间戳。注意，它总是使用当前日期；和你可以覆盖的那种默认值不一样。

`DateField.auto_now_add`

当对象第一次被创建时自动设置当前时间。用于创建时间的时间戳. 它总是使用当前日期；和你可以覆盖的那种默认值不一样。

该字段默认对应的表单控件是一个 `TextInput`. 在管理员站点添加了一个JavaScript写的日历控件，和一个“Today"的快捷按钮.包含了一个额外的 `invalid_date` 错误消息键.

`auto_now_add`, `auto_now`, and `default` 这些设置是相互排斥的. 他们之间的任何组合将会发生错误的结果.

Note

在目前的实现中，设置 `auto_now` 或者 `auto_now_add` 为 `True` 将为让这个字段同时得到 `editable=False` 和 `blank=True` 这两个设置.

Note

`auto_now` and `auto_now_add` 这两个设置会在对象创建或更新的时刻,总是使用*default timezone*(默认时区)的日期. 如果你不想这样，你可以考虑一下简单地使用你自己的默认调用或者重写 `save()` (在save()函数里自己添加保存时间的机制. 译者注)而不是使用 `auto_now` or `auto_now_add`; 或者使用 `DateTimeField` 字段类来替换 `DateField` 并且在给用户呈现时间的时候,决定如何处理从datetime到date的转换.

DateTimeField

class `DateTimeField` ([*auto_now=False*, *auto_now_add=False*, ***options*])

它是通过Python `datetime.datetime` 实例表示的日期和时间. 携带了跟 `DateField` 一样的额外参数.

该字段默认对应的表单控件是一个单个的 `TextInput` (单文本输入框). 管理界面是使用两个带有 JavaScript控件的? `TextInput` ?文本框.

DecimalField

class `DecimalField` (*max_digits=None*, *decimal_places=None*[, ***options*])

用python中 `Decimal` 的一个实例来表示十进制浮点数. 有两个 必须的参数:

`DecimalField.max_digits`

位数总数，包括小数点后的位数。该值必须大于等于 `decimal_places` .

`DecimalField.decimal_places`

小数点后的数字数量

例如,要保存最大为 `999` 并有两位小数的数字,你应该使用:

```
models.DecimalField(..., max_digits=5, decimal_places=2)
```

而要存储那些将近10亿，并且要求达到小数点后十位精度的数字:

```
models.DecimalField(..., max_digits=19, decimal_places=10)
```

该字段默认的窗体组件是 `TextInput` .

Note

想获取更多关于 `FloatField` 和 `DecimalField` 差异, 请参照 *FloatField vs. DecimalField*.

DurationField

New in Django 1.8.

class `DurationField` ([***options*])

用作存储一段时间的字段类型 - 类似Python中的 `timedelta` . 当数据库使用的是PostgreSQL, 该数据类型使用的是一个 `interval` 而在Oracle上，则使用的是 `INTERVAL DAY(9) TO SECOND(6)` . Otherwise a `bigint` of microseconds is used.

注意

`DurationField` 的算数运算在多数情况下可以很好的工作。?然而，在除了PostgreSQL之外的其他数据库中, 将 `DurationField` 与 `DateTimeField` 的实例比较则不会得到正确的结果。

EmailField

class `EmailField` ([*max_length=254*, ***options*])

一个 `CharField` 用来检查输入的email地址是否合法。它使用 `EmailValidator` 来验证输入合法性。

Changed in Django 1.8:

默认最大长度 `max_length` 从75增加到254以符合RFC3696/5321标准。

FileField

class `FileField` ([*upload_to=None*, *max_length=100*, ***options*])

一个上传文件的字段。

注意

FileField字段不支持 `primary_key` 和 `unique` 参数，如果使用会生成? `TypeError` 错误

有两个可选参数：

`FileField.upload_to`

Changed in Django 1.7:

在旧版本Django中， `upload_to` ?属性是必须要有的;

一个本地文件系统的路径，它将附加到 `MEDIA_ROOT` 设置的后面来确定 `url` 属性的值。

这个路径可能会包含一个 `strftime()` 格式串,并且会在文件上传时被替换为 实际的date/time作为文件路径 (这样上传的文件就不会塞满你指定的文件夹了).

它还可以是一个可调用对象如函数，将调用它来获取上传路径，包括文件名。这个可调用对象必须接受两个参数，并且返回一个Unix 风格的路径（带有前向/）给存储系统。将传递的两个参数为：

Argument	Description
`instance`	`FileField` 定义所在的模型的实例。更准确地说，就是当前文件的所在的那个实例。大部分情况下，这个实例将还没有保存到数据库中，若它用到默认的 `AutoField` 字段，它的主键字段还可能没有值。
`filename`	The filename that was originally given to the file. This may or may not be taken into account when determining the final destination path.

`FileField.storage`

一个Storage 对象，用于你的文件的存取。参见管理文件 获取如何提供这个对象的细节。

这个字段的默认表单Widget 是 `ClearableFileInput` 。

在模型中调用 `FileField` 或 `ImageField` (见下方) 需如下几步：

1. 在你的settings文件中，你必须要定义 `MEDIA_ROOT` 作为Django存储上传文件的路径(从性能上考虑，这些文件不能存在数据库中。) 定义一个 `MEDIA_URL` 作为基础的URL或者目录。确保这个目录可以被web server使用的账户写入。
2. 在模型中添加 `FileField` 或 `ImageField` 字段，定义 `upload_to` 参数，内容是 `MEDIA_ROOT` 的子目录，用来存放上传的文件。
3. 数据库中存放的仅是这个文件的路径 （相对于 `MEDIA_ROOT`). 你很可能会想用由Django提供的便利的 `url` 属性。比如说，如果你的 `ImageField` 命名为 `mug_shot` , 你可以在template中用 ``获得你照片的绝对路径。

例如，如果你的 `MEDIA_ROOT` 设定为 `'/home/media'` ，并且 `upload_to` 设定为 `'photos/%Y/%m/%d'` 。 `upload_to` 的 `'%Y/%m/%d'` 被 `strftime()` 所格式化； `'%Y'` 将会被格式化为一个四位数的年份， `'%m'` 被格式化为一个两位数的月份 `'%d'` 是两位数日份。如果你在Jan.15.2007上传了一个文件，它将被保存在 `/home/media/photos/2007/01/15` 目录下.

如果你想获得上传文件的存盘文件名，或者是文件大小，你可以分别使用 `name` 和 `size` 属性； 更多可用属性及方法信息，请参见 `File` 类索引 和 *Managing files* 主题指导.

Note

保存的文件作为模型存储在数据库中的一部分，所以在磁盘上使用的实际的文件名在模型保存完毕之前是不可靠的。

上传的文件对应的URL可以通过使用 `url` 属性获得. 在内部，它会调用 ? `Storage` 类下的 `url()` 方法.

值得注意的是，无论你在任何时候处理上传文件的需求，你都应该密切关注你的文件将被上传到哪里，上传的文件类型，以避免安全漏洞。认证所有上传文件 以确保那些上传的文件是你所认为的文件。例如，如果你盲目的允许其他人在无需认证的情况下上传文件至你的web服务器的root目录中，那么别人可以上传一个CGI或者PHP脚本然后通过访问一个你网站的URL来执行这个脚本。所以，不要允许这种事情发生。

甚至是上传HTML文件也值得注意，它可以通过浏览器（虽然不是服务器）执行，也可以引发相当于是XSS或者CSRF攻击的安全威胁。

`FileField` 实例将会在你的数据库中创建一个默认最大长度为100字符的 `varchar` 列。就像其他的fields一样，你可以用 `max_length` 参数改变最大长度的值.

FileField 和FieldFile

class `FieldFile` [source]

当你添加 `FileField` 到你的模型中时，你实际上会获得一个 `FieldFile` 的实例来替代将要访问的文件。 除了继承至 `django.core.files.File` 的功能外，这个类还有其他属性和方法可以用于访问文件:

`FieldFile.url`

通过潜在 `Storage` 类的 `url()` 方法可以只读地访问文件的URL。

`FieldFile.open` (*mode='rb'*)[source]

该方法像标准的Python `open()` 方法,并可通过? `mode` 参数设置打开模式.

`FieldFile.close` ()[source]

该方法像标准的Python `file.close()` 方法,并关闭相关文件.

`FieldFile.save` (*name*, *content*, *save=True*)[source]

这个方法会将文件名以及文件内容传递到字段的storage类中,并将模型字段与保存好的文件关联. 如果想要手动关联文件数据到你的模型中的 `FileField` 实例,则 `save()` 方法总是用来保存该数据.

方法接受两个必选参数: `name` 文件名, 和 `content` 文件内容.可选参数 `save` 控制模型实例在关联的文件被修改时是否保存.默认为 `True` .

注意参数 `content` 应该是 `django.core.files.File` 的一个实例, 而不是Python内建的File对象.你可以用如下方法从一个已经存在的Python文件对象来构建 `File` :

```
from django.core.files import File
# Open an existing file using Python's built-in open()
f = open('/tmp/hello.world')
myfile = File(f)
```

或者，你可以像下面的一样从一个python字符串中构建

```
from django.core.files.base import ContentFile
myfile = ContentFile("hello world")
```

更多信息, 请参见 *Managing files*.

`FieldFile.delete` (*save=True*)[source]

删除与此实例关联的文件，并清除该字段的所有属性。注意： 如果它碰巧是开放的调用 `delete()` 方法 时，此方法将关闭该文件。

模型实例 `save` 的文件与此字段关联的可选 保存 参数控件已被删除。默认值为 `True` 。

注意，model删除的时候，与之关联的文件并不会被删除。如果你要把文件也清理掉，你需要自己处理。

FilePathField

class `FilePathField` (*path=None*[, *match=None*, *recursive=False*, *max_length=100*, ***options*])

一个? `CharField` ，内容只限于文件系统内特定目录下的文件名。有三个参数, 其中第一个是 必需的:

`FilePathField.path`

必填。这个 `FilePathField` 应该得到其选择的目录的绝对文件系统路径。例如: `"/home/images"` .

`FilePathField.match`

可选的. `FilePathField` 将会作为一个正则表达式来匹配文件名。但请注意正则表达式将将被作用于基本文件名，而不是完整路径。例如: `"foo.*.txt$"` , 将会匹配到一个名叫 `foo23.txt` 的文件，但不匹配到 `bar.txt` 或者 `foo23.png` .

`FilePathField.recursive`

可选的. `True` 或 `False` .默认是 `False` .声明是否包含所有子目录的 路径

`FilePathField.allow_files`

可选的. `True` 或 `False` .默认是 `True` .声明是否包含指定位置的文件。该参数或 `allow_folders` 中必须有一个为 `True` .

`FilePathField.allow_folders`

是可选的.输入 `True` 或者 `False` .默认值为 `False` .声明是否包含指定位置的文件夹。该参数或 `allow_files` 中必须有一个为 `True` .

当然，这些参数可以同时使用。

有一点需要提醒的是 `match` 只匹配基本文件名（base filename）, 而不是整个文件路径（full path）. 例如:

```
FilePathField(path="/home/images", match="foo.*", recursive=True
)
```

...将匹配 `/home/images/foo.png` 而不是 `/home/images/foo/bar.png` 因为只允许 匹配 基本文件名(`foo.png` 和 `bar.png`).

`FilePathField` 实例被创建在您的数据库为 `varchar` 列默认最大长度为 100 个字符。作为与其他字段，您可以更改使用的 `max_length` 最大长度。

FloatField

class `FloatField` ([***options*])

用Python的一个 `float` 实例来表示一个浮点数.

该字段的默认组件是一个 `TextInput` .

`FloatField` 与 `DecimalField`

有时候 `FloatField` 类会和 `DecimalField` 类发生混淆. 虽然它们表示都表示实数，但是二者表示数字的方式不一样。 `FloatField` 使用的是Python内部的 `float` 类型, 而 `DecimalField` 使用的是Python的 `Decimal` 类型. 想要了解更多二者的差别, 可以查看Python文档中的 `decimal` 模块.

ImageField

class `ImageField` ([*upload_to=None*, *height_field=None*, *width_field=None*, *max_length=100*, ***options*])

继承了 `FileField` 的所有属性和方法, 但还对上传的对象进行校验，确保它是个有效的image.

除了从 `FileField` 继承来的属性外， `ImageField` 还有 `宽` 和 `高` 属性。

为了更便捷的去用那些属性值, `ImageField` 有两个额外的可选参数

`ImageField.height_field`

该属性的设定会在模型实例保存时,自动填充图片的高度.

`ImageField.width_field`

该属性的设定会在模型实例保存时,自动填充图片的宽度.

ImageField字段需要调用Pillow 库.

`ImageField` 会创建在你的数据库中 和 `varchar` 一样,默认最大长度为100和其他字段一样, 你可以使用 `max_length` 参数来设置默认文件最大值.

此字段的默认表单工具是 `ClearableFileInput` .

IntegerField

class `IntegerField` ([***options*])

一个整数。在Django所支持的所有数据库中，从 `-2147483648` 到 `2147483647` 范围内的值是合法的。默认的表单输入工具是 `TextInput` .

IPAddressField

class `IPAddressField` ([***options*])

Deprecated since version 1.7: 该字段已废弃，从1.7开始支持 `GenericIPAddressField` .

IP地址，会自动格式化（例如：“192.0.2.30”）。默认表单控件为 `TextInput`.

GenericIPAddressField

class `GenericIPAddressField` ([*protocol=both*, *unpack_ipv4=False*, ***options*])

一个 IPv4 或 IPv6 地址, 字符串格式 (例如 `192.0.2.30` 或 `2a02:42fe::4`). 这个字段的默认表单小部件是一个 `TextInput` .

IPv6 地址会根据 **RFC 4291** 章节 2.2所规范, 包括该章节中第三段的的IPv4格式建议, 就像 `::ffff:192.0.2.0` 这样. 例如, `2001:0::0:01` 将会被规范成 `2001::1` , `::ffff:0a0a:0a0a` 被规范成 `::ffff:10.10.10.10` . 所有字符都会被转换成小写.

`GenericIPAddressField.protocol`

限制有效输入的协议类型. 允许的值是 `'both'` (默认值), `'IPv4'` 或 `'IPv6'` . 匹配不区分大小写.

`GenericIPAddressField.unpack_ipv4`

解析IPv4映射地址如 `::ffff:192.0.2.1` .如果启用该选项，则地址将被解析到 `192.0.2.1` .默认为禁用。只有当 `协议` 设置为 `'both'` 时才可以使用。

如果允许空白值，则必须允许null值，因为空白值存储为null。

NullBooleanField

class `NullBooleanField` ([***options*])

类似 `BooleanField` , 但是允许 `NULL` 作为一个选项.使用此代替 `null=True` 的 `BooleanField` 。此字段的默认表单widget为 `NullBooleanSelect` 。

PositiveIntegerField（正整数字段）

class `PositiveIntegerField` ([***options*])

类似 `IntegerField` , 但值必须是正数或者零(`0`). 从 `0` 到 `2147483647` 的值在Django支持的所有数据库中都是安全的。由于向后兼容性原因，接受值 `0` 。

PositiveSmallIntegerField

class `PositiveSmallIntegerField` ([***options*])

该模型字段类似 `PositiveIntegerField` , 但是只允许小于某一特定值（依据数据库类型而定）。从 `0` 到 `32767` 这个区间，对于Django所支持的所有数据库而言都是安全的。

SlugField

class `SlugField` ([*max_length=50*, ***options*])

Slug 是一个新闻术语（通常叫做短标题）。一个slug只能包含字母、数字、下划线或者是连字符，通常用来作为短标签。通常它们是用来放在URL里的。

像CharField一样，你可以指定 `max_length` （也请参阅该部分中的有关数据库可移植性的说明和 `max_length` ）。如果没有指定 `max_length` , Django将会默认长度为50。

Implies setting `Field.db_index` to `True` .

根据某些其他值的值自动预填充SlugField通常很有用。你可以在admin中使用 `prepopulated_fields` 自动执行此操作。

SmallIntegerField

class `SmallIntegerField` ([***options*])

与 `IntegerField` 这个字段类型很类似,不同的是SmallInteger类型只能在一个确定的范围内(数据库依赖)。对于django来讲，该字段值在 `-32768` 至 `32767` 这个范围内对所有可支持的数据库都是安全的。

TextField

class `TextField` ([***options*])

大文本字段。该模型默认的表单组件是 `Textarea` 。

Changed in Django 1.7:

如果你在这个字段类型中使用了 `max_length` 属性，它将会在渲染页面表单元素 `Textarea` 时候体现出来。但是并不会在model或者数据库级别强制性的限定字段长度。请使用 `CharField` 。

MySQL 用户

如果你将此字段用于MySQLdb 1.2.1p2和 `utf8_bin` 排序规则（这不是默认值），则需要注意一些问题。有关详细信息，请参阅*MySQL数据库注释*。

TimeField

class `TimeField` ([*auto_now=False*, *auto_now_add=False*, ***options*])

时间字段，和Python中 `datetime.time` 一样。接受与 `DateField` 相同的自动填充选项。

表单默认为 `TextInput` .输入框。Admin添加一些JavaScript快捷方式。

URLField

class `URLField` ([*max_length=200*, ***options*])

一个 `CharField` 类型的URL

此字段的默认表单widget为 `TextInput` 。

与所有 `CharField` 子类一样， `URLField` 接受可选的 `max_length` 参数。如果不指定 `max_length` ，则使用默认值200。

UUIDField

New in Django 1.8.

class `UUIDField` ([***options*])

一个用来存储UUID的字段。使用Python的 `UUID` 类。 当使用PostgreSQL数据库时，该字段类型对应的数据库中的数据类型是 `uuid` ，使用其他数据库时，数据库对应的是 `char(32)` 类型。

使用UUID类型相对于使用具有 `primary_key` 参数的 `AutoField` 类型是一个更好的解决方案。 数据库不会自动生成UUID，所以推荐使用 `default` 参数：

```
import uuid
from django.db import models

class MyUUIDModel(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    # other fields
```

注意：这里传递给default是一个可调用的对象（即一个省略了括号的方法），而不是传递一个 `UUID` 实例给 `default`

关系字段

Django 同样定义了一系列的字段来描述数据库之间的关联。

ForeignKey

class `ForeignKey` (*othermodel*[, ***options*])

多对一关系。需要一个位置参数：与该模型关联的类。

若要创建一个递归的关联 —— 对象与自己具有多对一的关系 —— 请使用 `models.ForeignKey('self')` 。

如果你需要关联到一个还没有定义的模型，你可以使用模型的名字而不用模型对象本身：

```
from django.db import models

class Car(models.Model):
    manufacturer = models.ForeignKey('Manufacturer')
    # ...

class Manufacturer(models.Model):
    # ...
    pass
```

若要引用在其它应用中定义的模型，你可以用带有完整标签名的模型来显式指定。例如，如果上面提到的 `Manufacturer` 模型是在一个名为 `production` 的应用中定义的，你应该这样使用它：

```
class Car(models.Model):
    manufacturer = models.ForeignKey('production.Manufacturer')
```

在解析两个应用之间具有相互依赖的导入时，这种引用将会很有帮助。

`ForeignKey` 会自动创建数据库索引。你可以通过设置 `db_index` 为 `False` 来取消。如果你创建外键是为了一致性而不是用来Join，或者如果你将创建其它索引例如部分或多列索引，你也许想要避免索引的开销。

警告

不建议从一个没有迁移的应用中创建一个 `ForeignKey` 到一个具有迁移的应用。更多详细信息，请参阅依赖性文档。

数据库中的表示

在幕后，Django 会在字段名上添加 `"_id"` 来创建数据库中的列名。在上面的例子中， `Car` 模型的数据库表将会拥有一个 `manufacturer_id` 列。（你可以通过显式指定 `db_column` ?改变它）。但是，你的代码永远不应该处理数据库中的列名称，除非你需要编写自定义的SQL。你应该永远只处理你的模型对象中的字段名称。

参数

`ForeignKey` 接受额外的参数集 —— 全都是可选的 —— 它们定义关联关系如何工作的细节。

`ForeignKey.limit_choices_to`

当这个字段使用 `模型表单` 或者Admin 渲染时（默认情况下，查询集中的所有对象都可以使用），为这个字段设置一个可用的选项。它可以是一个字典、一个 `Q` 对象或者一个返回字典或 `Q` 对象的可调用对象。

例如：

```
staff_member = models.ForeignKey(User, limit_choices_to={'is_staff': True})
```

将使得 `模型表单` 中对应的字段只列出 `is_staff=True` 的 `Users` 。这在Django 的Admin 中也可能有用处。

可调用对象的形式同样非常有用，比如与Python 的 `datetime` 模块一起使用来限制选择的时间范围。例如：

```
def limit_pub_date_choices():
    return {'pub_date__lte': datetime.date.utcnow()}

limit_choices_to = limit_pub_date_choices
```

如果 `limit_choices_to` 自己本身是或者返回一个用于复杂查询的 `Q 对象` ，当字段没有在模型的 `ModelAdmin` 中的 `raw_id_fields` 列出时，它将只会影响Admin中的可用的选项。

Changed in Django 1.7:

以前的Django 版本不允许传递一个可调用的对象给 `limit_choices_to` 。

注

如果 `limit_choices_to` 使用可调用对象，这个可调用对象将在每次创建一个新表单的时候都调用。它还可能在一个模型校验的时候调用，例如被管理命令或者Admin。Admin 多次构造查询集来验证表单在各种边缘情况下的输入，所以你的可调用对象可能调用多次。

`ForeignKey.related_name`

这个名称用于让关联的对象反查到源对象。它还是 `related_query_name` 的默认值（关联的模型进行反向过滤时使用的名称）。完整的解释和示例参见关联对象的文档。注意，当你为抽象模型定义关联关系的时，必须设置这个参数的值；而且当你这么做的时候需要用到一些特殊语法。

如果你不想让Django 创建一个反向关联，请设置 `related_name` 为 `'+'` 或者以 `'+'` 结尾。 例如，下面这行将确定 `User` 模型将不会有到这个模型的返回关联：

```
user = models.ForeignKey(User, related_name='+')
```

`ForeignKey.related_query_name`

这个名称用于目标模型的反向过滤。如果设置了 `related_name` ，则默认为它的值，否则默认值为模型的名称：

```
# Declare the ForeignKey with related_query_name
class Tag(models.Model):
    article = models.ForeignKey(Article, related_name="tags", re
lated_query_name="tag")
    name = models.CharField(max_length=255)

# That's now the name of the reverse filter
Article.objects.filter(tag__name="important")
```

`ForeignKey.to_field`

关联到的关联对象的字段名称。默认地，Django 使用关联对象的主键。

`ForeignKey.db_constraint`

控制是否在数据库中为这个外键创建约束。默认值为 `True` ，而且这几乎一定是你想要的效果；设置成 `False` 对数据的完整性来说是很糟糕的。即便如此，有一些场景你也许想要这么设置：

- 你有遗留的无效数据。
- 你正在对数据库缩容。

如果被设置成 `False` ，访问一个不存在的关联对象将抛出 `DoesNotExist` 异常。

`ForeignKey.on_delete`

当一个 `ForeignKey` 引用的对象被删除时，Django 默认模拟SQL 的 `ON DELETE CASCADE` 的约束行为，并且删除包含该 `ForeignKey` 的对象。这种行为可以通过设置 `on_delete` 参数来改变。例如，如果你有一个可以为空的 `ForeignKey` ，在其引用的对象被删除的时你想把这个ForeignKey 设置为空：

```
user = models.ForeignKey(User, blank=True, null=True, on_delete=
models.SET_NULL)
```

`on_delete` 在 `django.db.models` 中可以找到的值有：

- `CASCADE`

  级联删除；默认值。

- `PROTECT`

  抛出 `ProtectedError` 以阻止被引用对象的删除，它是 `django.db.IntegrityError` 的一个子类。

- `SET_NULL`

  把 `ForeignKey` 设置为null； `null` 参数为 `True` 时才可以这样做。

- `SET_DEFAULT`

  `ForeignKey` 值设置成它的默认值；此时必须设置 `ForeignKey` 的default参数。

- `SET` ()

  设置 `ForeignKey` 为传递给 `SET()` 的值，如果传递的是一个可调用对象，则为调用后的结果。在大部分情形下，传递一个可调用对象用于避免models.py 在导入时执行查询：

  ```
  from django.conf import settings
      from django.contrib.auth import get_user_model
      from django.db import models

      def get_sentinel_user():
          return get_user_model().objects.get_or_create(userna
  me='deleted')[0]

      class MyModel(models.Model):
          user = models.ForeignKey(settings.AUTH_USER_MODEL,
                                   on_delete=models.SET(get_se
  ntinel_user))
  ```

- `DO_NOTHING`

  不采取任何动作。如果你的数据库后端强制引用完整性，它将引发一个 `IntegrityError` ，除非你手动添加一个 `ON DELETE` 约束给数据库自动（可能要用到*初始化的SQL*）。

`ForeignKey.swappable`

New in Django 1.7.

控制迁移框架的的重复行为如果该 `ForeignKey` 指向一个可切换的模型。如果它是默认值 `True` ，那么如果 `ForeignKey` 指向的模型与 `settings.AUTH_USER_MODEL` 匹配（或其它可切换的模型），则保存在迁移中的关联关系将使用对setting 中引用而不是直接对模型的引用。

只有当你确定你的模型将永远指向切换后的模型 —— 例如如果它是专门为你的自定义用户模型设计的模型时，你才会想将它设置成 `False` 。

设置为 `False` 并不表示你可以引用可切换的模型即使在它被切换出去之后 —— `False` 只是表示生成的迁移中ForeignKey 将始终引用你指定的准确模型（所以，如果用户试图允许一个你不支持的User 模型时将会失败）。

如果有疑问，请保留它的默认值 `True` 。

`ForeignKey.allow_unsaved_instance_assignment`

New in Django 1.8:

添加这个标志是为了向后兼容，因为老版本的Django 始终允许赋值未保存的模型实例。

Django 阻止未保存的模型实例被分配给一个 `ForeignKey` 字段以防止意味的数据丢失（当保存一个模型实例时，未保存的外键将默默忽略）。

如果你需要允许赋值未保存的实例且不关心数据的丢失（例如你不会保存对象到数据库），你可以通过创建这个字段的子类并设置其 `allow_unsaved_instance_assignment` 属性为 `True` 来关闭这个检查。例如：

```
class UnsavedForeignKey(models.ForeignKey):
    # A ForeignKey which can point to an unsaved object
    allow_unsaved_instance_assignment = True

class Book(models.Model):
    author = UnsavedForeignKey(Author)
```

ManyToManyField

class `ManyToManyField` (*othermodel*[, ***options*])

一个多对多关联。要求一个关键字参数：与该模型关联的类，与 `ForeignKey` 的工作方式完全一样，包括递归关系 和惰性关系。

关联的对象可以通过字段的 `RelatedManager` 添加、删除和创建。

警告

不建议从一个没有迁移的应用中创建一个 `ManyToManyField` 到一个具有迁移的应用。 更多细节参见依赖性文档。

数据库中的表示

在幕后，Django 创建一个中间表来表示多对多关系。默认情况下，这张中间表的名称使用多对多字段的名称和包含这张表的模型的名称生成。因为某些数据库支持的表的名字的长度有限制，这些标的名称将自动截短到64 个字符并加上一个唯一性的哈希值。这意味着，你看的表的名称可能类似 `author_books_9cdf4` ；这再正常不过了。你可以使用 `db_table` 选项手工提供中间表的名称。

参数

`ManyToManyField` 接收一个参数集来控制关联关系的功能 —— 它们都是可选的。

`ManyToManyField.related_name`

与 `ForeignKey.related_name` 相同。

`ManyToManyField.related_query_name`

与 `ForeignKey.related_query_name` 相同。

`ManyToManyField.limit_choices_to`

与 `ForeignKey.limit_choices_to` 相同。

`limit_choices_to` 对于使用 `through` 参数自定义中间表的 `ManyToManyField` 不生效。

`ManyToManyField.symmetrical`

只用于与自身进行关联的ManyToManyField。例如下面的模型：

```
from django.db import models

class Person(models.Model):
    friends = models.ManyToManyField("self")
```

当Django 处理这个模型的时候，它定义该模型具有一个与自身具有多对多关联的 `ManyToManyField`，因此它不会向 `Person` 类添加 `person_set` 属性。Django 将假定这个 `ManyToManyField` 字段是对称的 —— 如果我是你的朋友，那么你也是我的朋友。

如果你希望与 `self` 进行多对多关联的关系不具有对称性，可以设置 `symmetrical` 为 `False`。这会强制让Django 添加一个描述器给反向的关联关系，以使得 `ManyToManyField` 的关联关系不是对称的。

`ManyToManyField.through`

Django 会自动创建一个表来管理多对多关系。不过，如果你希望手动指定中介表，可以使用 `through` 选项来指定Django 模型来表示你想要使用的中介表。

这个选项最常见的使用场景是当你想要关联额外的数据到多对多关联关系的时候。

如果你没有显式指定 `through` 的模型，仍然会有一个隐式的 `through` 模型类，你可以用它来直接访问对应的表示关联关系的数据库表。它由三个字段来链接模型。

如果源模型和目标不同，则生成以下字段：

- `id`：关系的主键。
- `<containing_model>_id`：声明 `ManyToManyField` 字段的模型的 `id`。
- `<other_model>_id`：`ManyToManyField` 字段指向的模型的 `id`。

如果 `ManyToManyField` 的源模型和目标模型相同，则生成以下字段：

- `id` ：关系的主键。
- `from_<model>_id` ：源模型实例的 `id` 。
- `to_<model>_id` ：目标模型实例的 `id` 。

这个类可以让一个给定的模型像普通的模型那样查询与之相关联的记录。

`ManyToManyField.through_fields`

New in Django 1.7.

只能在指定了自定义中间模型的时候使用。 Django 一般情况会自动决定使用中间模型的哪些字段来建立多对多关联。但是，考虑如下模型：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=50)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership
', through_fields=('group', 'person'))

class Membership(models.Model):
    group = models.ForeignKey(Group)
    person = models.ForeignKey(Person)
    inviter = models.ForeignKey(Person, related_name="membership
_invites")
    invite_reason = models.CharField(max_length=64)
```

`Membership` 有两个 外键指向 `Person` （ `person` 和 `inviter` ），这使得关联关系含混不清并让Django 不知道使用哪一个。在这种情况下，你必须使用 `through_fields` 明确指定Django 应该使用哪些外键，就像上面例子一样。

`through_fields` 接收一个二元组 `('field1', 'field2')` ，其中 `field1` 为指向定义 `ManyToManyField` 字段的模型的外键名称（本例中为 `group` ）， `field2` 为指向目标模型的外键的名称（本例中为 `person` ）。

当中间模型具有多个外键指向多对多关联关系模型中的任何一个（或两个），你必须 指定 `through_fields` 。这通用适用于递归的关联关系，当用到中间模型而有多个外键指向该模型时，或当你想显式指定Django 应该用到的两个字段时。

递归的关联关系使用的中间模型始终定义为非对称的，也就是 `symmetrical=False` —— 所以具有源和目标的概念。这种情况下， `'field1'` 将作为管理关系的源，而 `'field2'` 作为目标。

`ManyToManyField.db_table`

为存储多对多数据而创建的表的名称。如果没有提供，Django 将基于定义关联关系的模型和字段假设一个默认的名称。

`ManyToManyField.db_constraint`

控制中间表中的外键是否创建约束。默认为 `True` ，而且这是几乎就是你想要的；设置为 `False` 对数据完整性将非常糟糕。下面是你可能需要这样设置的一些场景：

- 你具有不合法的遗留数据。
- 你正在对数据库缩容。

不可以同时传递 `db_constraint` 和 `through` 。

`ManyToManyField.swappable`

New in Django 1.7.

控制 `ManyToManyField` 指向一个可切换的模型时迁移框架的行为。如果它是默认值 `True` ，那么如果 `ManyToManyField` 指向的模型与 `settings.AUTH_USER_MODEL` 匹配（或其它可切换的模型），则保存在迁移中的关联关系将使用对setting 中引用而不是直接对模型的引用。

只有当你确定你的模型将永远指向切换后的模型 —— 例如如果它是专门为你的自定义用户模型设计的模型时，你才会想将它设置成 `False` 。

如果不确定，请将它保留为 `True` 。

`ManyToManyField.allow_unsaved_instance_assignment`

New in Django 1.8.

与 `ForeignKey.allow_unsaved_instance_assignment` 的工作方式类似。

`ManyToManyField` 不支持 `validators` 。

`null` 不生效，因为无法在数据库层次要求关联关系。

OneToOneField

class `OneToOneField` (*othermodel*[, *parent_link=False*, ***options*])

一对一关联关系。概念上讲，这个字段很像是 `ForeignKey` 设置了 `unique=True` ，不同的是它会直接返回关系另一边的单个对象。

它最主要的用途是作为扩展自另外一个模型的主键；例如，多表继承就是通过对子模型添加一个隐式的一对一关联关系到父模型实现的。

需要一个位置参数：与该模型关联的类。 它的工作方式与 `ForeignKey` 完全一致，包括所有与递归关系和惰性关系相关的选项。

如果你没有指定 `OneToOneField` 的 `related_name` 参数，Django 将使用当前模型的小写的名称作为默认值。

例如下面的例子：

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    supervisor = models.OneToOneField(settings.AUTH_USER_MODEL,
related_name='supervisor_of')
```

你将使得 `User` 模型具有以下属性：

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'myspecialuser')
True
>>> hasattr(user, 'supervisor_of')
True
```

当反向访问关联关系时，如果关联的对象不存在对应的实例，则抛出 `DoesNotExist` 异常。例如，如果一个User没有 `MySpecialUser` 指定的supervisor：

```
>>> user.supervisor_of
Traceback (most recent call last):
    ...
DoesNotExist: User matching query does not exist.
```

另外， `OneToOneField` 除了接收 `ForeignKey` 接收的所有额外的参数之外，还有另外一个参数：

`OneToOneField.parent_link`

当它为 `True` 并在继承自另一个具体模型 的模型中使用时，表示该字段应该用于反查的父类的链接，而不是在子类化时隐式创建的 `OneToOneField` 。

`OneToOneField` 的使用示例参见*One-to-one* 关联关系。

Field API 参考

class `Field`

`Field` 是一个抽象的类, 用来代表数据库中的表的一列. Django 使用这些fields 去创建表 (`db_type()`), 去建立Python中的类型和数据库中类型的映射关系 (`get_prep_value()`) 反之亦然 (`from_db_value()`), 并且实现*Lookup API reference* (`get_prep_lookup()`).

field 是不同Django版本API中最根本的部分，尤其是 `models` and `querysets` .

在模型中，一个字段被实例化为类的属性，并表现为一个特定的表的列，详情查看*Models*. 它具有 `null` 和 `唯一` 等属性，以及Django用于将字段值映射到数据库特定值的方法。

`字段` 是 `RegisterLookupMixin` 的子类，因此可以在其上注册 `Transform` 和 `Lookup` `QuerySet` s（例如 `field_name__exact ="foo"` ）。默认情况下，所有内置查找都已注册。

Django的所有内建字段，如 `CharField` 都是 `Field` 的特定实现。如果您需要自定义字段，则可以将任何内置字段子类化，也可以从头开始写入 `字段` 。无论哪种情况，请参阅编写自定义模型字段。

`description`

字段的详细说明，例如用于 `django.contrib.admindocs` 应用程序。

描述可以是以下形式：

```
description = _("String (up to %(max_length)s)")
```

其中参数从字段的 `__ dict __` 插入。

To map a `Field` to a database-specific type, Django exposes two methods:

`get_internal_type` ()

返回一个字符串，命名此字段以用于后端特定用途。默认情况下，它返回类名。

有关自定义字段中的用法，请参见模拟内置字段类型。

`db_type` (*connection*)

返回 `字段` 的数据库列数据类型，同时考虑 `连接` 。

有关自定义字段中的用法，请参见自定义数据库类型。

有三种主要情况，Django需要与数据库后端和字段交互：

- 当它查询数据库（Python值 转为 数据库后端值）
- 当它从数据库加载数据（数据库后端值 转为 Python值）
- 当它保存到数据库（Python值 转为 数据库后端值）

查询时，使用 `get_db_prep_value()` 和 `get_prep_value()` ：

`get_prep_value` (*value*)

`value` 是模型属性的当前值，方法应以已准备好用作查询中的参数的格式返回数据。

有关使用方式，请参阅将*Python*对象转换为查询值。

`get_db_prep_value` (*value*, *connection*, *prepared=False*)

将 `值` 转换为后端特定值。如果 `prepared = True` 和 `get_prep_value()` if 为 `False` ，则默认返回 `值` 。

有关用法，请参见将查询值转换为数据库值。

加载数据时，使用 `from_db_value()` ：

`from_db_value` (*value*, *expression*, *connection*, *context*)

New in Django 1.8.

将数据库返回的值转换为Python对象。它与 `get_prep_value()` 相反。

此方法不用于大多数内置字段，因为数据库后端已返回正确的Python类型，或后端本身执行转换。

有关用法，请参见将值转换为*Python*对象。

注意

出于性能原因， `from_db_value` 在不需要它的字段（所有Django字段）上不实现为无操作。因此，您不能在定义中调用 `super` 。

保存时，使用 `pre_save()` 和 `get_db_prep_save()`

`get_db_prep_save` (*value*, *connection*)

与 `get_db_prep_value()` 相同，但在字段值必须保存到数据库时调用。默认返回 `get_db_prep_value()` 。

`pre_save` (*model_instance*, *add*)

在 `get_db_prep_save()` 之前调用方法以在保存之前准备值（例如，对于 `DateField.auto_now` ）。

`model_instance` 是此字段所属的实例， `add` 是实例是否第一次保存到数据库。

它应该返回此字段的 `model_instance` 适当属性的值。属性名称位于 `self.attname` （这是由 `Field` 设置）。

有关使用情况，请参阅保存前预处理值。

当在字段上使用查找时，该值可能需要“准备”。Django公开了两种方法：

`get_prep_lookup` (*lookup_type*, *value*)

在用于查找之前，准备 `values` 到数据库。The `lookup_type` will be one of the valid Django filter lookups: `"exact"` , `"iexact"` , `"contains"` , `"icontains"` , `"gt"` , `"gte"` , `"lt"` , `"lte"` , `"in"` , `"startswith"` , `"istartswith"` , `"endswith"` , `"iendswith"` , `"range"` , `"year"` , `"month"` , `"day"` , `"isnull"` , `"search"` , `"regex"` , and `"iregex"` .

New in Django 1.7.

如果你使用自定义查找，则 `lookup_type` 可以是在字段中注册的任何 `lookup_name` 。

有关用法，请参见准备在数据库查找中使用的值。

`get_db_prep_lookup` (*lookup_type*, *value*, *connection*, *prepared=False*)

类似于 `get_db_prep_value()` ，但用于执行查找。

与 `get_db_prep_value()` 一样，将用于查询的特定连接作为 `connection` 传递。此外， `prepared` 描述该值是否已经用 `get_prep_lookup()` 准备好。

字段经常从不同的类型接收它们的值，从序列化或从表单。

`to_python` (*value*)

改变这个值为正确的python对象。它作为 `value_to_string()` 的反向操作，也在 `clean()` 中调用。

有关用法，请参见将值转换为*Python*对象。

除了保存到数据库，该字段还需要知道如何序列化其值：

`value_to_string` (*obj*)

将 `obj` 转换为字符串。用于序列化字段的值。

有关用法，请参见转换字段数据以进行序列化。

当使用 模型 表单 时， `Field` 需要知道应由哪个表单字段表示：

`formfield` (*form_class=None*, *choices_form_class=None*, ***kwargs*)

返回 `ModelForm` 的此字段的默认 `django.forms.Field` 。

By default, if both `form_class` and `choices_form_class` are `None` , it uses `CharField` ; if `choices_form_class` is given, it returns `TypedChoiceField` .

有关用法，请参见为模型字段指定表单字段。

`deconstruct` ()

New in Django 1.7.

返回具有足够信息的4元组，以重新创建字段：

1. 模型上字段的名称。
2. 字段的导入路径（例如 `“django.db.models.IntegerField”` ）。这应该是兼容的版本，所以不要太具体可能会更好。
3. 位置参数列表。
4. 关键字参数的字典。

必须将此方法添加到1.7之前的字段，才能使用迁移迁移其数据。

Field属性参考

New in Django 1.8.

每个 `字段` 实例包含几个允许内省其行为的属性。Use these attributes instead of `isinstance` checks when you need to write code that depends on a field's functionality. 这些属性可与*Model._meta API*一起使用，以缩小特定字段类型的搜索范围。自定义模型字段应实现这些标志。

Attributes for fields

`Field.auto_created`

布尔标志，指示是否自动创建字段，例如模型继承使用的 `OneToOneField` 。

`Field.concrete`

布尔标志，指示字段是否具有与其相关联的数据库列。

`Field.hidden`

Boolean flag that indicates if a field is used to back another non-hidden field's functionality (e.g. the `content_type` and `object_id` fields that make up a `GenericForeignKey`). The `hidden` flag is used to distinguish what constitutes the public subset of fields on the model from all the fields on the model.

Note

`Options.get_fields()` 默认排除隐藏字段。传入 `include_hidden = True` 可返回结果中的隐藏字段。

`Field.is_relation`

布尔标志，指示字段是否包含对一个或多个其他模型的功能的引用（例如 `ForeignKey` ， `ManyToManyField` ， `OneToOneField` 等） 。

`Field.model`

返回定义字段的模型。如果在模型的超类上定义了字段，则 `模型` 将引用超类，而不是实例的类。

具有关系的字段的属性

这些属性用于查询关系的基数和其他详细信息。这些属性存在于所有字段上；但是，如果字段是关系类型（ `Field.is_relation=True` ） ，则它们只有布尔值（而不是 `None` ） 。

`Field.many_to_many`

如果字段具有多对多关系，则布尔标志为 `True` ；否则为 `False` 。The only field included with Django where this is `True` is `ManyToManyField` .

`Field.many_to_one`

如果字段具有多对一关系，例如 `ForeignKey` ，则布尔标志为 `True` ；否则为 `False` 。

`Field.one_to_many`

如果字段具有一对多关系（例如 `GenericRelation` 或 `ForeignKey` 的反向），则 `True` 的布尔标志；否则为 `False` 。

`Field.one_to_one`

如果字段具有一对一关系，例如 `OneToOneField` ，则布尔标志为 `True` ；否则为 `False` 。

`Field.related_model`

指向字段涉及的模型。例如， `ForeignKey(Author)` 中的 `Author` 。如果字段具有通用关系（例如 `GenericForeignKey` 或 `GenericRelation` ），则 `related_model` 将为 `None` 。

> 译者：Django 文档协作翻译小组，原文：Field types。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

模型元选项

这篇文档阐述了所有可用的元选项，你可以在你模型的Meta类中设置他们。

可用的元选项

abstract

Options.abstract

如果 abstract = True， 就表示模型是 抽象基类 (abstract base class).

app_label

Options.app_label

如果你的模型定义在默认的 models.py 之外(比如，你现在用的模型在 myapp.models 子模块当中)，你必须告诉 Django 该模型属于哪个应用：

```
app_label = 'myapp'
```

```
Django 1.7中新增：

一个应用中，定义在models 模块以外的模型，不再需要app_label。
```

db_table

Options.db_table

该模型所用的数据表的名称：

```
db_table = 'music_album'
```

数据表名称

为了节省时间，Django 会根据模型类的名称和包含它的app名称自动指定数据表名称，一个模型的数据表名称，由这个模型的“应用标签”（在 manage.py startapp中使用的名称）之间加上下划线组成。

举个例子， bookstore应用(使用 manage.py startapp bookstore 创建)，里面有个名为 Book的模型，那数据表的名称就是 bookstore_book 。

使用 Meta类中的 db_table 参数来覆写数据表的名称。

数据表名称可以是 SQL 保留字，也可以包含不允许出现在 Python 变量中的特殊字符，这是因为 Django 会自动给列名和表名添加引号。

> 在 MySQL中使用小写字母为表命名
>
> 当你通过db_table覆写表名称时，强烈推荐使用小写字母给表命名，特别是如果你用了MySQL作为后端。详见MySQL注意事项 。
>
> Oracle中表名称的引号处理
>
> 为了遵从Oracle中30个字符的限制，以及一些常见的约定，Django会缩短表的名称，而且会把它全部转为大写。在db_table的值外面加上引号来避免这种情况：
>
> ```
> db_table = '"name_left_in_lowercase"'
> ```
>
> 这种带引号的名称也可以用于Django所支持的其他数据库后端，但是除了Oracle，引号不起任何作用。详见 Oracle 注意事项 。

db_tablespace

Options.db_tablespace

当前模型所使用的数据库表空间 的名字。默认值是项目设置中的DEFAULT_TABLESPACE，如果它存在的话。如果后端并不支持表空间，这个选项可以忽略。

default_related_name

Options.default_related_name

```
Django 1.8中新增：
```

这个名字会默认被用于一个关联对象到当前对象的关系。默认为 _set。

由于一个字段的反转名称应该是唯一的，当你给你的模型设计子类时，要格外小心。为了规避名称冲突，名称的一部分应该含有'%(app_label)s'和'%(model_name)s'，它们会被应用标签的名称和模型的名称替换，二者都是小写的。详见抽象模型的关联名称。

get_latest_by

Options.get_latest_by

模型中某个可排序的字段的名称，比如DateField、DateTimeField或者IntegerField。它指定了Manager的latest()和earliest()中使用的默认字段。

例如：

```
get_latest_by = "order_date"
```

详见latest() 文档。

managed

Options.managed

默认为True，意思是Django在migrate命令中创建合适的数据表，并且会在 flush 管理命令中移除它们。换句话说，Django会管理这些数据表的生命周期。

如果是False，Django 就不会为当前模型创建和删除数据表。如果当前模型表示一个已经存在的，通过其它方法建立的数据库视图或者数据表，这会相当有用。这是设置为managed=False时唯一的不同之处。. 模型处理的其它任何方面都和平常一样。这包括：

- 如果你不声明它的话，会向你的模型中添加一个自增主键。为了避免给后面的代码读者带来混乱，强烈推荐你在使用未被管理的模型时，指定数据表中所有的列。
- 如果一个带有managed=False的模型含有指向其他未被管理模型的ManyToManyField，那么多对多连接的中介表也不会被创建。但是，一个被管理模型和一个未被管理模型之间的中介表会被创建。

如果你需要修改这一默认行为，创建中介表作为显式的模型（设置为managed），并且使用ManyToManyField.through为你的自定义模型创建关联。

对于带有managed=False的模型的测试，你要确保在测试启动时建立正确的表。

如果你对修改模型类在Python层面的行为感兴趣，你可以设置 managed=False ，并且创建一个已经存在模型的部分。但是这种情况下使用代理模型才是更好的方法。

order_with_respect_to

Options.order_with_respect_to

按照给定的字段把这个对象标记为”可排序的“。这一属性通常用到关联对象上面，使它在父对象中有序。比如，如果Answer和 Question相关联，一个问题有至少一个答案，并且答案的顺序非常重要，你可以这样做：

```
from django.db import models

class Question(models.Model):
    text = models.TextField()
    # ...

class Answer(models.Model):
    question = models.ForeignKey(Question)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

当order_with_respect_to 设置之后，模型会提供两个用于设置和获取关联对象顺序的方法：get_RELATED_order() 和set_RELATED_order()，其中RELATED是小写的模型名称。例如，假设一个 Question 对象有很多相关联的Answer对象，返回的列表中含有相关联Answer对象的主键：

```
>>> question = Question.objects.get(id=1)
>>> question.get_answer_order()
[1, 2, 3]
```

与Question对象相关联的Answer对象的顺序，可以通过传入一格包含Answer 主键的列表来设置：

```
>>> question.set_answer_order([3, 1, 2])
```

相关联的对象也有两个方法， get_next_in_order() 和get_previous_in_order()，用于按照合适的顺序访问它们。假设Answer对象按照 id来排序：

```
>>> answer = Answer.objects.get(id=2)
>>> answer.get_next_in_order()
<Answer: 3>
>>> answer.get_previous_in_order()
<Answer: 1>
```

> 修改 order_with_respect_to
>
> order_with_respect_to属性会添加一个额外的字段（/数据表中的列）叫做_order，所以如果你在首次迁移之后添加或者修改了order_with_respect_to属性，要确保执行和应用了合适的迁移操作。

ordering

Options.ordering

对象默认的顺序，获取一个对象的列表时使用：

```
ordering = ['-order_date']
```

它是一个字符串的列表或元组。每个字符串是一个字段名，前面带有可选的“-”前缀表示倒序。前面没有“-”的字段表示正序。使用"?"来表示随机排序。

例如，要按照pub_date字段的正序排序，这样写：

```
ordering = ['pub_date']
```

按照pub_date字段的倒序排序，这样写：

```
ordering = ['-pub_date']
```

先按照pub_date的倒序排序，再按照 author 的正序排序，这样写：

```
ordering = ['-pub_date', 'author']
```

> 警告
>
> 排序并不是没有任何代价的操作。你向ordering属性添加的每个字段都会产生你数据库的开销。你添加的每个外键也会隐式包含它的默认顺序。

permissions

Options.permissions

设置创建对象时权限表中额外的权限。增加、删除和修改权限会自动为每个模型创建。这个例子指定了一种额外的权限，can_deliver_pizzas：

```
permissions = (("can_deliver_pizzas", "Can deliver pizzas"),)
```

它是一个包含二元组的元组或者列表，格式为 (permission_code, human_readable_permission_name)。

default_permissions

Options.default_permissions

```
Django 1.7中新增：
```

默认为('add', 'change', 'delete')。你可以自定义这个列表，比如，如果你的应用不需要默认权限中的任何一项，可以把它设置成空列表。在模型被migrate命令创建之前，这个属性必须被指定，以防一些遗漏的属性被创建。

proxy

Options.proxy

如果proxy = True, 作为该模型子类的另一个模型会被视为代理模型。

select_on_save

Options.select_on_save

该选项决定了Django是否采用1.6之前的 django.db.models.Model.save()算法。旧的算法使用SELECT来判断是否存在需要更新的行。而新式的算法直接尝试使用UPDATE。在一些小概率的情况中，一个已存在的行的UPDATE操作并不对Django可见。比如PostgreSQL的ON UPDATE触发器会返回NULL。这种情况下，新式的算法会在最后执行 INSERT 操作，即使这一行已经在数据库中存在。

通常这个属性不需要设置。默认为False。

关于旧式和新式两种算法，请参见django.db.models.Model.save()。

unique_together

Options.unique_together

用来设置的不重复的字段组合：

```
unique_together = (("driver", "restaurant"),)
```

它是一个元组的元组，组合起来的时候必须是唯一的。它在Django后台中被使用，在数据库层上约束数据(比如，在 CREATE TABLE 语句中包含 UNIQUE语句)。

为了方便起见，处理单一字段的集合时，unique_together 可以是一维的元组：

```
unique_together = ("driver", "restaurant")
```

ManyToManyField不能包含在unique_together中。（这意味着什么并不清楚！）如果你需要验证ManyToManyField关联的唯一性，试着使用信号或者显式的贯穿模型(explicit through model)。

```
Django 1.7中修改：
当unique_together的约束被违反时，模型校验期间会抛出ValidationError异常
。
```

index_together

Options.index_together

用来设置带有索引的字段组合：

```
index_together = [
    ["pub_date", "deadline"],
]
```

列表中的字段将会建立索引（例如，会在CREATE INDEX语句中被使用）。

```
Django 1.7中修改：
```

为了方便起见，处理单一字段的集合时，index_together可以是一个一维的列表。

```
index_together = ["pub_date", "deadline"]
```

verbose_name

Options.verbose_name

对象的一个易于理解的名称，为单数：

```
verbose_name = "pizza"
```

如果此项没有设置，Django会把类名拆分开来作为自述名，比如CamelCase 会变成camel case，

verbose_name_plural

Options.verbose_name_plural

该对象复数形式的名称：

```
verbose_name_plural = "stories"
```

如果此项没有设置，Django 会使用 verbose_name + "s"。

Model 类参考

这篇文档覆盖 `Model` 类的特性。关于模型的更多信息，参见Model 完全参考指南。

属性

objects

`Model.objects`

每个非抽象的 `Model` 类必须给自己添加一个 `Manager` 实例。Django 确保在你的模型类中至少有一个默认的 `Manager` 。如果你没有添加自己的 `Manager` ，Django 将添加一个属性 `objects` ，它包含默认的 `Manager` 实例。如果你添加自己的 `Manager` 实例的属性，默认值则不会出现。思考一下下面的例子：

```
from django.db import models

class Person(models.Model):
    # Add manager with another name
    people = models.Manager()
```

关于模型管理器的更多信息，参见Managers 和 Retrieving objects。

> 译者：Django 文档协作翻译小组，原文：Model class。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

查询集

执行查询

一旦你建立好数据模型之后，django会自动生成一套数据库抽象的API，可以让你执行增删改查的操作。这篇文档阐述了如何使用这些API。关于所有模型检索选项的详细内容，请见数据模型参考。

在整个文档（以及参考）中，我们会大量使用下面的模型，它构成了一个博客应用。

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):              # __unicode__ on Python 2
        return self.headline
```

创建对象

为了把数据库表中的数据表示成python对象，django使用一种直观的方式：一个模型类代表数据库的一个表，一个模型的实例代表数据库表中的一条特定的记录。

使用关键词参数实例化一个对象来创建它，然后调用**save()**把它保存到数据库中。

假设模型存放于文件**mysite/blog/models.py**中，下面是一个例子：

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatle
s news.')
>>> b.save()
```

上面的代码在背后执行了sql的**INSERT**操作。在你显式调用**save()**之前，django不会访问数据库。

save()方法没有返回值。

> 请参见
>
> **save()**方法带有一些高级选项，它们没有在这里给出，完整的细节请见**save()**文档。
>
> 如果你想只用一条语句创建并保存一个对象，使用**create()**方法。

保存对象的改动

调用**save()**方法，来保存已经存在于数据库中的对象的改动。

假设一个**Blog**的实例**b5**已经被保存在数据库中，这个例子更改了它的名字，并且在数据库中更新它的记录：

```
>>> b5.name = 'New name'
>>> b5.save()
```

上面的代码在背后执行了sql的**UPDATE**操作。在你显式调用**save()**之前，django不会访问数据库。

保存**ForeignKey**和**ManyToManyField**字段

更新**ForeignKey**字段的方式和保存普通字段相同--只是简单地把一个类型正确的对象赋值到字段中。下面的例子更新了**Entry**类的实例**entry**的**blog**属性，假设**Entry**的一个合适的实例以及**Blog**已经保存在数据库中（我们可以像下面那样获取他们）：

```
>>> from blog.models import Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

更新**ManyToManyField**的方式有一些不同--使用字段的**add()**方法来增加关系的记录。这个例子向**entry**对象添加**Author**类的实例**joe**：

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

为了在一条语句中，向**ManyToManyField**添加多条记录，可以在调用**add()**方法时传入多个参数，像这样：

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django将会在你添加错误类型的对象时抛出异常。

获取对象

通过模型中的**Manager**构造一个**QuertSet**，来从你的数据库中获取对象。

QuerySet表示你数据库中取出来的一个对象的集合。它可以含有零个、一个或者多个过滤器，过滤器根据所给的参数限制查询结果的范围。在sql的角度，**QuerySet**和**SELECT**命令等价，过滤器是像**WHERE**和**LIMIT**一样的限制子句。

你可以从模型的**Manager**那里取得**QuerySet**。每个模型都至少有一个**Manager**，它通常命名为**objects**。通过模型类直接访问它，像这样：

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

> 注意
>
> 管理器通常只可以通过模型类来访问，不可以通过模型实例来访问。这是为了强制区分表级别和记录级别的操作。

对于一个模型来说，**Manager**是**QuerySet**的主要来源。例如， **Blog.objects.all()**会返回持有数据库中所有**Blog**对象的一个**QuerySet**。

获取所有对象

获取一个表中所有对象的最简单的方式是全部获取。使用**Manager**的**all()**方法：

```
>>> all_entries = Entry.objects.all()
```

all()方法返回包含数据库中所有对象的**QuerySet**。

使用过滤器获取特定对象

all()方法返回的结果集中包含全部对象，但是更普遍的情况是你需要获取完整集合的一个子集。

要创建这样一个子集，需要精炼上面的结果集，增加一些过滤器作为条件。两个最普遍的途径是：

filter(kwargs)** 返回一个包含对象的集合，它们满足参数中所给的条件。

exclude(kwargs)** 返回一个包含对象的集合，它们不满足参数中所给的条件。

查询参数（上面函数定义中的****kwargs**）需要满足特定的格式，字段检索一节中会提到。

举个例子，要获取年份为2006的所有文章的结果集，可以这样使用**filter()**方法：

```
Entry.objects.filter(pub_date__year=2006)
```

在默认的管理器类中，它相当于：

```
Entry.objects.all().filter(pub_date__year=2006)
```

链式过滤

QuerySet的精炼结果还是**QuerySet**，所以你可以把精炼用的语句组合到一起，像这样：

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 30)
... )
```

最开始的**QuerySet**包含数据库中的所有对象，之后增加一个过滤器去掉一部分，在之后又是另外一个过滤器。最后的结果的一个**QuerySet**，包含所有标题以”word“开头的记录，并且日期是2005年一月，日为当天的值。

过滤后的结果集是独立的

每次你筛选一个结果集，得到的都是全新的另一个结果集，它和之前的结果集之间没有任何绑定关系。每次筛选都会创建一个独立的结果集，可以被存储及反复使用。

例如：

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

这三个 QuerySets 是不同的。 第一个 QuerySet 包含大标题以"What"开头的所有记录。第二个则是第一个的子集，用一个附加的条件排除了出版日期 pub_date 是今天的记录。 第三个也是第一个的子集，它只保留出版日期 pub_date 是今天的记录。 最初的 QuerySet (q1) 没有受到筛选的影响。

查询集是延迟的

QuerySets 是惰性的 -- 创建 QuerySet 的动作不涉及任何数据库操作。你可以一直添加过滤器，在这个过程中，Django 不会执行任何数据库查询，除非 QuerySet 被执行. 看看下面这个例子:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.now())
>>> q = q.exclude(body_text__icontains="food")
>>> print q
```

虽然上面的代码看上去象是三个数据库操作，但实际上只在最后一行 (print q) 执行了一次数据库操作，。一般情况下， QuerySet 不能从数据库中主动地获得数据，得被动地由你来请求。对 QuerySet 求值就意味着 Django 会访问数据库。想了解对查询集何时求值，请查看 何时对查询集求值 (When QuerySets are evaluated).

其他查询集方法

大多数情况使用 all(), filter() 和 exclude() 就足够了。 但也有一些不常用的；请查看 查询API参考 (QuerySet API Reference) 中完整的 QuerySet 方法列表。

限制查询集范围

可以用 python 的数组切片语法来限制你的 QuerySet 以得到一部分结果。它等价于SQL中的 LIMIT 和 OFFSET 。

例如，下面的这个例子返回前五个对象 (LIMIT 5):

```
>>> Entry.objects.all()[:5]
```

这个例子返回第六到第十之间的对象 (OFFSET 5 LIMIT 5):

```
>>> Entry.objects.all()[5:10]
```

Django 不支持对查询集做负数索引 (例如 Entry.objects.all()[-1]) 。

一般来说，对 QuerySet 切片会返回新的 QuerySet -- 这个过程中不会对运行查询。不过也有例外，如果你在切片时使用了 "step" 参数，查询集就会被求值，就在数据库中运行查询。举个例子，使用下面这个这个查询集返回前十个对象中的偶数次对象，就会运行数据库查询:

```
>>> Entry.objects.all()[:10:2]
```

要检索单独的对象，而非列表 (比如 SELECT foo FROM bar LIMIT 1)，可以直接使用索引来代替切片。举个例子，下面这段代码将返回大标题排序后的第一条记录 Entry:

```
>>> Entry.objects.order_by('headline')[0]
```

大约等价于：

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

要注意的是：如果找不到符合条件的对象，第一种方法会抛出 IndexError ，而第二种方法会抛出 DoesNotExist。 详看 get() 。

字段筛选条件

字段筛选条件就是 SQL 语句中的 WHERE 从句。就是 Django 中的 QuerySet 的 filter(), exclude() 和 get() 方法中的关键字参数。

筛选条件的形式是 field__lookuptype=value 。 (注意：这里是双下划线)。例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

大体可以翻译为如下的 SQL 语句:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

这是怎么办到的？

Python 允许函式接受任意多 name-value 形式的参数，并在运行时才确定name和value的值。详情请参阅官方Python教程中的 关键字参数(Keyword Arguments)。

如果你传递了一个无效的关键字参数，会抛出 TypeError 导常。

数据库 API 支持24种查询类型；可以在 字段筛选参考(field lookup reference) 查看详细的列表。为了给您一个直观的认识，这里我们列出一些常用的查询类型：

exact

"exact" 匹配。例如：

```
>>> Entry.objects.get(headline__exact="Man bites dog")
```

会生成如下的 SQL 语句：

```
SELECT ... WHERE headline = 'Man bites dog';
```

如果你没有提供查询类型 -- 也就是说关键字参数中没有双下划线，那么查询类型就会被指定为 exact。

举个例子，这两个语句是相等的：

```
>>> Blog.objects.get(id__exact=14)  # Explicit form
>>> Blog.objects.get(id=14)         # __exact is implied
```

这样做很方便，因为 exact 是最常用的。

iexact

忽略大小写的匹配。所以下面的这个查询:

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

会匹配标题是 "Beatles Blog", "beatles blog", 甚至 "BeAtlES blOG" 的 Blog

contains

大小写敏感的模糊匹配。 例如：

```
Entry.objects.get(headline__contains='Lennon')
```

大体可以翻译为如下的 SQL：

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

要注意这段代码匹配大标题 'Today Lennon honored' ，而不能匹配 'today lennon honored'。

它也有一个忽略大小写的版本，就是 icontains。

startswith, endswith

分别匹配开头和结尾，同样也有忽略大小写的版本 istartswith 和 iendswith。 再强调一次，这仅仅是简短介绍。完整的参考请参见 字段筛选条件参考(field lookup reference)。

跨关系查询

Django 提供了一种直观而高效的方式在查询(lookups)中表示关联关系，它能自动确认 SQL JOIN 联系。要做跨关系查询，就使用两个下划线来链接模型(model)间关联字段的名称，直到最终链接到你想要的 model 为止。

这个例子检索所有关联 Blog 的 name 值为 'Beatles Blog' 的所有 Entry 对象：

```
>>> Entry.objects.filter(blog__name__exact='Beatles Blog')
```

跨关系的筛选条件可以一直延展。

关系也是可逆的。可以在目标 model 上使用源 model 名称的小写形式得到反向关联。

下面这个例子检索至少关联一个 Entry 且大标题 headline 包含 'Lennon' 的所有 Blog 对象：

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

如果在某个关联 model 中找不到符合过滤条件的对象，Django 将视它为一个空的 (所有的值都是 NULL), 但是可用的对象。这意味着不会有异常抛出，在这个例子中：

```
Blog.objects.filter(entry__author__name='Lennon')
```

(假设关联到 Author 类), 如果没有哪个 author 与 entry 相关联，Django 会认为它没有 name 属性，而不会因为不存在 author 抛出异常。通常来说，这正是你所希望的机制。唯一的例外是使用 isnull 的情况。如下:

```
Blog.objects.filter(entry__author__name__isnull=True)
```

这段代码会得到 author 的 name 为空的 Blog 或 entry 的 author为空的 Blog。 如果不嫌麻烦，可以这样写：

```
Blog.objects.filter (entry__author__isnull=False,
        entry__author__name__isnull=True)
```

跨一对多／多对多关系(Spanning multi-valued relationships)

这部分是Django 1.0中新增的： 请查看版本记录 如果你的过滤是基于 ManyToManyField 或是逆向 ForeignKeyField 的，你可能会对下面这两种情况感兴趣。回顾 Blog/Entry 的关系(Blog 到 Entry 是一对多关系)，如果要查找这样的 blog：它关联一个大标题包含"Lennon"，且在2008年出版的 entry ；或者要查找这样的 blogs：它关联一个大标题包含"Lennon"的 entry ，同时它又关联另外一个在2008年出版的 entry 。因为一个 Blog 会关联多个的Entry，所以上述两种情况在现实应用中是很有可能出现的。

同样的情形也出现在 ManyToManyField 上。例如，如果 Entry 有一个 ManyToManyField 字段，名字是 tags，我们想得到 tags 是"music"和"bands"的 entries，或者我们想得到包含名为"music" 的标签而状态是"public"的 entry。

针对这两种情况，Django 用一种很方便的方式来使用 filter() 和 exclude()。对于包含在同一个 filter() 中的筛选条件，查询集要同时满足所有筛选条件。而对于连续的 filter() ，查询集的范围是依次限定的。但对于跨一对多／多对多关系查询来说，在第二种情况下，筛选条件针对的是主 model 所有的关联对象，而不是被前面的 filter() 过滤后的关联对象。

这听起来会让人迷糊，举个例子会讲得更清楚。要检索这样的 blog：它要关系一个大标题中含有 "Lennon" 并且在2008年出版的 entry (这个 entry 同时满足这两个条件)，可以这样写：

```
Blog.objects.filter(entry__headline__contains='Lennon',
        entry__pub_date__year=2008)
```

要检索另外一种 blog：它关联一个大标题含有"Lennon"的 entry ，又关联一个在2008年出版的 entry （一个 entry 的大标题含有 Lennon，同一个或另一个 entry 是在2008年出版的）。可以这样写：

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
        entry__pub_date__year=2008)
```

在第二个例子中，第一个过滤器(filter)先检索与符合条件的 entry 的相关联的所有 blogs。第二个过滤器在此基础上从这些 blogs 中检索与第二种 entry 也相关联的 blog。第二个过滤器选择的 entry 可能与第一个过滤器所选择的完全相同，也可能不同。 因为过滤项过滤的是 Blog，而不是 Entry。

上述原则同样适用于 exclude()：一个单独 exclude() 中的所有筛选条件都是作用于同一个实例 (如果这些条件都是针对同一个一对多／多对多的关系)。连续的 filter() 或 exclude() 却根据同样的筛选条件，作用于不同的关联对象。

在过滤器中引用 model 中的字段(Filters can reference fields on the model)

这部分是 Django 1.1 新增的: 请查看版本记录 在上面所有的例子中，我们构造的过滤器都只是将字段值与某个常量做比较。如果我们要对两个字段的值做比较，那该怎么做呢？

Django 提供 F() 来做这样的比较。F() 的实例可以在查询中引用字段，来比较同一个 model 实例中两个不同字段的值。

例如：要查询回复数(comments)大于广播数(pingbacks)的博文(blog entries)，可以构造一个 F() 对象在查询中引用评论数量：

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_pingbacks__lt=F('n_comments'))
```

Django 支持 F() 对象之间以及 F() 对象和常数之间的加减乘除和取模的操作。例如，要找到广播数等于评论数两倍的博文，可以这样修改查询语句：

```
>>> Entry.objects.filter(n_pingbacks__lt=F('n_comments') * 2)
```

要查找阅读数量小于评论数与广播数之和的博文，查询如下:

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingb
acks'))
```

你也可以在 F() 对象中使用两个下划线做跨关系查询。F() 对象使用两个下划线引入必要的关联对象。例如，要查询博客(blog)名称与作者(author)名称相同的博文(entry)，查询就可以这样写：

```
>>> Entry.objects.filter(author__name=F('blog__name'))
```

主键查询的简捷方式

为使用方便考虑，Django 用 pk 代表主键"primary key"。

以 Blog 为例, 主键是 id 字段，所以下面三个语句都是等价的：

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

pk 对 __exact 查询同样有效，任何查询项都可以用 pk 来构造基于主键的查询：

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

pk 查询也可以跨关系，下面三个语句是等价的：

```
>>> Entry.objects.filter(blog__id__exact=3) # Explicit form
>>> Entry.objects.filter(blog__id=3)        # __exact is implied
>>> Entry.objects.filter(blog__pk=3)        # __pk implies __id_
_exact
```

在LIKE语句中转义百分号%和下划线_

字段筛选条件相当于 LIKE SQL 语句 (iexact, contains, icontains, startswith, istartswith, endswith 和 iendswith) ，它会自动转义两个特殊符号 -- 百分号%和下划线。*(在 LIKE 语句中，百分号%表示多字符匹配，而下划线表示单字符匹配。)*

这就意味着我们可以直接使用这两个字符，而不用考虑他们的 SQL 语义。例如，要查询大标题中含有一个百分号%的 entry：

```
>>> Entry.objects.filter(headline__contains='%')
```

Django 会处理转义；最终的 SQL 看起来会是这样：

```
SELECT ... WHERE headline LIKE '%\%%';
```

下划线_和百分号%的处理方式相同，Django 都会自动转义。

缓存和查询

每个 QuerySet 都包含一个缓存，以减少对数据库的访问。要编写高效代码，就要理解缓存是如何工作的。

一个 QuerySet 时刚刚创建的时候，缓存是空的。 QuerySet 第一次运行时，会执行数据库查询，接下来 Django 就在 QuerySet 的缓存中保存查询的结果，并根据请求返回这些结果(比如，后面再次调用这个 QuerySet 的时候)。再次运行 QuerySet 时就会重用这些缓存结果。

要牢住上面所说的缓存行为，否则在使用 QuerySet 时可能会给你造成不小的麻烦。例如，创建下面两个 QuerySet ，并对它们求值，然后释放：

```
>>> print [e.headline for e in Entry.objects.all()]
>>> print [e.pub_date for e in Entry.objects.all()]
```

这就意味着相同的数据库查询将执行两次，事实上读取了两次数据库。而且，这两次读出来的列表可能并不完全相同，因为存在这种可能：在两次读取之间，某个 Entry 被添加到数据库中，或是被删除了。

要避免这个问题，只要简单地保存 QuerySet 然后重用即可：

```
>>> queryset = Poll.objects.all()
>>> print [p.headline for p in queryset] # Evaluate the query se
t.
>>> print [p.pub_date for p in queryset] # Re-use the cache from
 the evaluation.
```

用 Q 对象实现复杂查找 (Complex lookups with Q objects)

在 filter() 等函式中关键字参数彼此之间都是 "AND" 关系。如果你要执行更复杂的查询(比如，实现筛选条件的 OR 关系)，可以使用 Q 对象。

Q 对象(django.db.models.Q)是用来封装一组查询关键字的对象。这里提到的查询关键字请查看上面的 "Field lookups"。

例如，下面这个 Q 对象封装了一个单独的 LIKE 查询：

```
Q(question__startswith='What')
```

Q 对象可以用 & 和 | 运算符进行连接。当某个操作连接两个 Q 对象时，就会产生一个新的等价的 Q 对象。

例如，下面这段语句就产生了一个 Q ，这是用 "OR" 关系连接的两个 "question__startswith" 查询：

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

上面的例子等价于下面的 SQL WHERE 从句：

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

你可以用 & 和 | 连接任意多的 Q 对象，而且可以用括号分组。Q 对象也可以用 ~ 操作取反，而且普通查询和取反查询(NOT)可以连接在一起使用：

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

每种查询函式(比如 filter(), exclude(), get()) 除了能接收关键字参数以外，也能以位置参数的形式接受一个或多个 Q 对象。如果你给查询函式传递了多个 Q 对象，那么它们彼此间都是 "AND" 关系。例如：

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... 大体可以翻译为下面的 SQL：

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

查找函式可以混用 Q 对象和关键字参数。查询函式的所有参数(Q 关系和关键字参数) 都是 "AND" 关系。但是，如果参数中有 Q 对象，它必须排在所有的关键字参数之前。例如：

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who')
```

... 是一个有效的查询。但下面这个查询虽然看上去和前者等价：

```
# INVALID QUERY
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))
```

... 但这个查询却是无效的。

> 参见
>
> 在 Django 的单元测试 OR查询实例(OR lookups examples) 中展示了 Q 的用例。

对象比较

要比较两个对象，就和 Python 一样，使用双等号运算符：==。实际上比较的是两个 model 的主键值。

以上面的 Entry 为例，下面两个语句是等价的：

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

如果 model 的主键名称不是 id，也没关系。Django 会自动比较主键的值，而不管他们的名称是什么。例如，如果一个 model 的主键字段名称是 name，那么下面两个语句是等价的：

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

对象删除

删除方法就是 delete()。它运行时立即删除对象而不返回任何值。例如：

```
e.delete()
```

你也可以一次性删除多个对象。每个 QuerySet 都有一个 delete() 方法，它一次性删除 QuerySet 中所有的对象。

例如，下面的代码将删除 pub_date 是2005年的 Entry 对象：

```
Entry.objects.filter(pub_date__year=2005).delete()
```

要牢记这一点：无论在什么情况下，QuerySet 中的 delete() 方法都只使用一条 SQL 语句一次性删除所有对象，而并不是分别删除每个对象。如果你想使用在 model 中自定义的 delete() 方法，就要自行调用每个对象的delete 方法。(例如，遍历 QuerySet，在每个对象上调用 delete()方法)，而不是使用 QuerySet 中的 delete()方法。

在 Django 删除对象时，会模仿 SQL 约束 ON DELETE CASCADE 的行为，换句话说，删除一个对象时也会删除与它相关联的外键对象。例如：

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

要注意的是： delete() 方法是 QuerySet 上的方法，但并不适用于 Manager 本身。这是一种保护机制，是为了避免意外地调用 Entry.objects.delete() 方法导致 所有的记录被误删除。如果你确认要删除所有的对象，那么你必须显式地调用：

```
Entry.objects.all().delete()
```

一次更新多个对象 (Updating multiple objects at once)

这部分是 Django 1.0 中新增的： 请查看版本文档 有时你想对 QuerySet 中的所有对象，一次更新某个字段的值。这个要求可以用 update() 方法完成。例如：

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Every
thing is the same')
```

这种方法仅适用于非关系字段和 ForeignKey 外键字段。更新非关系字段时，传入的值应该是一个常量。更新 ForeignKey 字段时，传入的值应该是你想关联的那个类的某个实例。例如：

```
>>> b = Blog.objects.get(pk=1)

# Change every Entry so that it belongs to this Blog.
>>> Entry.objects.all().update(blog=b)
```

update() 方法也是即时生效，不返回任何值的(与 delete() 相似)。 在 QuerySet 进行更新时，唯一的限制就是一次只能更新一个数据表，就是当前 model 的主表。所以不要尝试更新关联表和与此类似的操作，因为这是不可能运行的。

要小心的是： update() 方法是直接翻译成一条 SQL 语句的。因此它是直接地一次完成所有更新。它不会调用你的 model 中的 save() 方法，也不会发出 pre_save 和 post_save 信号(这些信号在调用 save() 方法时产生)。如果你想保存 QuerySet 中的每个对象，并且调用每个对象各自的 save() 方法，那么你不必另外多写一个函式。只要遍历这些对象，依次调用 save() 方法即可：

```
for item in my_queryset:
    item.save()
```

这部分是在 Django 1.1 中新增的： 请查看版本文档 在调用 update 时可以使用 F() 对象 来把某个字段的值更新为另一个字段的值。这对于自增记数器是非常有用的。例如，给所有的博文 (entry) 的广播数 (pingback) 加一：

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

但是，与 F() 对象在查询时所不同的是，在filter 和 exclude子句中，你不能在 F() 对象中引入关联关系(NO-Join)，你只能引用当前 model 中要更新的字段。如果你在 F() 对象引入了Join 关系object，就会抛出 FieldError 异常：

```
# THIS WILL RAISE A FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

对象关联

当你定义在 model 定义关系时 (例如， ForeignKey, OneToOneField, 或 ManyToManyField)，model 的实例自带一套很方便的API以获取关联的对象。

以最上面的 models 为例，一个 Entry 对象 e 能通过 blog 属性获得相关联的 Blog 对象： e.blog。

(在场景背后，这个功能是由 Python 的 descriptors 实现的。如果你对此感兴趣，可以了解一下。)

Django 也提供反向获取关联对象的 API，就是由从被关联的对象得到其定义关系的主对象。例如，一个 Blog 类的实例 b 对象通过 entry_set 属性得到所有相关联的 Entry 对象列表: b.entry_set.all()。

这一节所有的例子都使用本页顶部所列出的 Blog, Author 和 Entry model。

一对多关系

正向

如果一个 model 有一个 ForeignKey字段，我们只要通过使用关联 model 的名称就可以得到相关联的外键对象。

例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

你可以设置和获得外键属性。正如你所期望的，改变外键的行为并不引发数据库操作，直到你调用 save()方法时，才会保存到数据库。例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

如果外键字段 ForeignKey 有一个 null=True 的设置(它允许外键接受空值 NULL)，你可以赋给它空值 None 。例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

在一对多关系中，第一次正向获取关联对象时，关联对象会被缓存。其后根据外键访问时这个实例，就会从缓存中获得它。例如：

```
>>> e = Entry.objects.get(id=2)
>>> print e.blog  # Hits the database to retrieve the associated
 Blog.
>>> print e.blog  # Doesn't hit the database; uses cached versio
n.
```

要注意的是，QuerySet 的 select_related() 方法提前将所有的一对多关系放入缓存中。例如：

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print e.blog  # Doesn't hit the database; uses cached versio
n.
>>> print e.blog  # Doesn't hit the database; uses cached versio
n.
```

逆向关联

如果 model 有一个 ForeignKey外键字段，那么外联 model 的实例可以通过访问 Manager 来得到所有相关联的源 model 的实例。默认情况下，这个 Manager 被命名为 FOO_set, 这里面的 FOO 就是源 model 的小写名称。这个 Manager 返回 QuerySets，它是可过滤和可操作的，在上面 "对象获取(Retrieving objects)" 有提及。

例如：

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blo
g.

# b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

你可以通过在 ForeignKey() 的定义中设置 related_name 的值来覆写 FOO_set 的名称。例如，如果 Entry model 中做一下更改： blog = ForeignKey(Blog, related_name='entries')，那么接下来就会如我们看到这般：

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

你不能在一个类当中访问 ForeignKey Manager ；而必须通过类的实例来访问：

```
>>> Blog.entry_set
Traceback:
    ...
AttributeError: "Manager must be accessed via instance".
```

除了在上面 "对象获取Retrieving objects" 一节中提到的 QuerySet 方法之外，ForeignKey Manager 还有如下一些方法。下面仅仅对它们做一个简短介绍，详情请查看 related objects reference。

`add(obj1, obj2, ...)`

将某个特定的 model 对象添加到被关联对象集合中。

`create(**kwargs)`

创建并保存一个新对象，然后将这个对象加被关联对象的集合中，然后返回这个新对象。

`remove(obj1, obj2, ...)`

将某个特定的对象从被关联对象集合中去除。

`clear()`

清空被关联对象集合。 想一次指定关联集合的成员，那么只要给关联集合分配一个可迭代的对象即可。它可以包含对象的实例，也可以只包含主键的值。例如：

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

在这个例子中，e1 和 e2 可以是完整的 Entry 实例，也可以是整型的主键值。

如果 clear() 方法是可用的，在迭代器(上例中就是一个列表)中的对象加入到 entry_set 之前，已存在于关联集合中的所有对象将被清空。如果 clear() 方法 不可用，原有的关联集合中的对象就不受影响，继续存在。

这一节提到的每一个 "reverse" 操作都是实时操作数据库的，每一个添加，创建，删除操作都会及时保存将结果保存到数据库中。

多对多关系

在多对多关系的任何一方都可以使用 API 访问相关联的另一方。多对多的 API 用起来和上面提到的 "逆向" 一对多关系关系非常相象。

唯一的差虽就在于属性的命名： ManyToManyField 所在的 model (为了方便，我称之为源model A) 使用字段本身的名称来访问关联对象；而被关联的另一方则使用 A 的小写名称加上 '_set' 后缀(这与逆向的一对多关系非常相象)。

下面这个例子会让人更容易理解：

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

与 ForeignKey 一样, ManyToManyField 也可以指定 related_name。在上面的例子中，如果 Entry 中的 ManyToManyField 指定 related_name='entries'，那么接下来每个 Author 实例的 entry_set 属性都被 entries 所代替。

一对一关系

相对于多对一关系而言，一对一关系不是非常简单的。如果你在 model 中定义了一个 OneToOneField 关系，那么你就可以用这个字段的名称做为属性来访问其所关联的对象。

例如：

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

与 "reverse" 查询不同的是，一对一关系的关联对象也可以访问 Manager 对象，但是这个 Manager 表现一个单独的对象，而不是一个列表：

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

如果一个空对象被赋予关联关系，Django 就会抛出一个 DoesNotExist 异常。

和你定义正向关联所用的方式一样，类的实例也可以赋予逆向关联方系：

```
e.entrydetail = ed
```

关系中的反向连接是如何做到的？

其他对象关系的映射(ORM)需要你在关联双方都定义关系。而 Django 的开发者则认为这违背了 DRY 原则 (Don't Repeat Yourself)，所以 Django 只需要你在一方定义关系即可。

但仅由一个 model 类并不能知道其他 model 类是如何与它关联的，除非是其他 model 也被载入，那么这是如何办到的？

答案就在于 INSTALLED_APPS 设置中。任何一个 model 在第一次调用时，Django 就会遍历所有的 INSTALLED_APPS 的所有 models，并且在内存中创建中必要的反向连接。本质上来说，INSTALLED_APPS 的作用之一就是确认 Django 完整的 model 范围。

在关联对象上的查询

包含关联对象的查询与包含普通字段值的查询都遵循相同的规则。为某个查询指定某个值的时候，你可以使用一个类实例，也可以使用对象的主键值。

例如，如果你有一个 Blog 对象 b ，它的 id=5, 下面三个查询是一样的：

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

直接使用**SQL**

如果你发现某个 SQL 查询用 Django 的数据库映射来处理会非常复杂的话，你可以使用直接写 SQL 来完成。

建议的方式是在你的 model 自定义方法或是自定义 model 的 manager 方法来运行查询。虽然 Django 不要求数据操作必须在 model 层中执行。但是把你的商业逻辑代码放在一个地方，从代码组织的角度来看，也是十分明智的。详情请查看 执行原生SQL查询(Performing raw SQL queries).

最后，要注意的是，Django的数据操作层仅仅是访问数据库的一个接口。你可以用其他的工具，编程语言，数据库框架来访问数据库。对你的数据库而言，没什么是非用 Django 不可的。

> 译者：Django 文档协作翻译小组，原文：Executing queries。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

QuerySet API参考

本文档描述了 `QuerySet` API的详细信息。它建立在模型和数据库查询指南的基础上，所以在阅读本文档之前，你也许需要首先阅读这两部分的文档。

本文档将通篇使用在数据库查询指南中用到的*Weblog* 模型的例子。

何时对查询集求值(要写到程序里的字段麻烦不要自作聪明翻译谢谢)

在内部，可以创建、过滤、切片和传递 `查询集` 而不用真实操作数据库。在你对查询集做求值之前，不会发生任何实际的数据库操作。

你可以使用下列方法对 `查询集` 求值：

- 迭代。 `queryset` 是可迭代的，它在首次迭代查询集时执行实际的数据库查询。例如， 下面的语句会将数据库中所有Entry 的headline 打印出来：

  ```
  for e in Entry.objects.all():
      print(e.headline)
  ```

  注意：不要使用上面的语句来验证在数据库中是否至少存在一条记录。使用 `exists()` 方法更高效。

- 切片。 正如在限制查询集中解释的那样， 可以使用Python 的序列切片语法对一个 `查询集` 进行分片。一个未求值的 `查询集` 进行切片通常返回另一个未求值的 `查询集` ，但是如果你使用切片的”step“参数，Django 将执行数据库查询并返回一个列表。对一个已经求值的 `查询集` 进行切片将返回一个列表。

  还要注意，虽然对未求值的 `查询集` 进行切片返回另一个未求值的 `查询集` ，但是却不可以进一步修改它了（例如，添加更多的Filter，或者修改排序的方式），因为这将不太好翻译成SQL而且含义也不清晰。

- 序列化/缓存。 序列化查询集的细节参见下面一节。本节提到它的目的是强调序列化将读取数据库。

- **repr()**。 当对 `查询集` 调用 `repr()` 时，将对它求值。这是为了在Python 交互式解释器中使用的方便，这样你可以在交互式使用这个API 时立即看到结果。

- **len()**。 当你对 `查询集` 调用 `len()` 时， 将对它求值。正如你期望的那样，返回一个查询结果集的长度。

注：如果你只需要知道集合中记录的个数（并不需要真实的对象），使用数据库层级的 `SELECT COUNT(*)` 计数将更加高效。为此，Django 提供了 一个 `count()` 方法.

- **list()**。 对 `查询集` 调用 `list()` 将强制对它求值。例如：

```
entry_list = list(Entry.objects.all())
```

- **bool()**。 测试一个 `查询集` 的布尔值，例如使用 `bool()` 、 `or` 、 `and` 或者 `if` 语句将导致查询集的执行。如果至少有一个记录，则 `查询集` 为 `True` ，否则为 `False` 。例如:

```
if Entry.objects.filter(headline="Test"):
   print("There is at least one Entry with the headline Test
")
```

注：如果你需要知道是否存在至少一条记录（而不需要真实的对象），使用 `exists()` 将更加高效。

Pickle 查询集

如果你 `Pickle` 一个 `查询集` ，它将在Pickle 之前强制将所有的结果加载到内存中。Pickle 通常用于缓存之前，并且当缓存的查询集重新加载时，你希望结果已经存在随时准备使用（从数据库读取耗费时间，就失去了缓存的目的）。这意味着当你Unpickle `查询集` 时，它包含Pickle 时的结果，而不是当前数据库中的结果。

如果此后你只想Pickle 必要的信息来从数据库重新创建 `查询集` ，可以Pickle `查询集` 的 `query` 属性。然后你可以使用类似下面的代码重新创建原始的 `查询集` （不用加载任何结果）：

```
>>> import pickle
>>> query = pickle.loads(s)     # Assuming 's' is the pickled st
ring.
>>> qs = MyModel.objects.all()
>>> qs.query = query            # Restore the original 'query'.
```

`query` 是一个不透明的对象。它表示查询的内部构造，不属于公开的API。然而，这里讲到的Pickle 和Unpickle 这个属性的内容是安全的（和完全支持的）。

不可以在不同版本之间共享Pickle 的结果

`查询集` 的Pickle 只能用于生成它们的Django 版本中。如果你使用Django 的版本N生成一个Pickle，不保证这个Pickle 在Django 的版本N+1 中可以读取。Pickle 不可用于归档的长期策略。

New in Django 1.8.

因为Pickle 兼容性的错误很难诊断例如产生损坏的对象，当你试图Unpickle 的查询集与Pickle 时的Django 版本不同，将引发一个 `RuntimeWarning` 。

查询集 API

下面是对于 `查询集` 的正式定义：

class `QuerySet` ([*model=None*, *query=None*, *using=None*])[source]

通常你在使用 `QuerySet` 时会以链式的*filter* 来使用。为了让这个能工作，大部分 `QuerySet` 方法返回新的QuerySet。这些方法在本节将详细讲述。

`QuerySet` 类具有两个公有属性用于内省：

`ordered` [source]

如果 `QuerySet` 是排好序的则为 `True` —— 例如有一个 `order_by()` 子句或者模型有默认的排序。否则为 `False` .

`db` [source]

如果现在执行，则返回将使用的数据库。

注

`QuerySet` 存在 `query` 参数是为了让具有特殊查询用途的子类如 `GeoQuerySet` 可以重新构造内部的查询状态。这个参数的值是查询状态的不透明的表示，不是一个公开的API。简而言之：如果你有疑问，那么你实际上不需要使用它。

返回新的查询集 的方法(要写到程序里的字段麻烦不要自作聪明翻译谢谢)

Django 提供了一系列 的 `QuerySet` 筛选方法，用于改变 `QuerySet` 返回的结果类型或者SQL查询执行的方式。

filter

`filter` (***kwargs*)

返回一个新的 `QuerySet` ，包含与给定的查询参数匹配的对象。

查找的参数（ `**kwargs` ）应该满足下文字段查找中的格式。在底层的SQL 语句中，多个参数通过 `AND` 连接。

如果你需要执行更复杂的查询（例如，使用 `OR` 语句查询），你可以使用 `Q 对象` 。

exclude

`exclude` (***kwargs*)

返回一个新的 `QuerySet` ，它包含不满足给定的查找参数的对象。

查找的参数（ `**kwargs` ）应该满足下文字段查找中的格式。 在底层的SQL 语句中，多个参数通过 `AND` 连接，然后所有的内容放入 `NOT()` 中。

下面的示例排除所有 `pub_date` 晚于2005-1-3 且 `headline` 为“Hello” 的记录：

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3), headline='Hello')
```

用SQL 语句，它等同于：

```
SELECT ...
WHERE NOT (pub_date > '2005-1-3' AND headline = 'Hello')
```

下面的示例排除所有 `pub_date` 晚于2005-1-3 或者headline 为“Hello” 的记录：

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3)).exclude(headline='Hello')
```

用SQL 语句，它等同于：

```
SELECT ...
WHERE NOT pub_date > '2005-1-3'
AND NOT headline = 'Hello'
```

注意，第二个示例更严格。

如果你需要执行更复杂的查询（例如，使用 `OR` 语句查询），你可以使用 `Q 对象` 。

annotate

`annotate` (**args*, ***kwargs*)

使用提供的查询表达式Annotate `查询集` 中的每个对象。查询表达式可以是一个简单的值、模型（或关联模型）字段的一个引用或对 `查询集` 中的对象一个聚合函数（平均值、和等）。

New in Django 1.8:

之前版本的Django 值允许聚合函数用作Annotation。现在可以使用各种表达式annotate 一个模型。

`annotate()` 的每个参数都是一个annotation，它将添加到返回的 `QuerySet` 中每个对象。

Django 提供的聚合函数在下文的聚合函数文档中讲述。

关键字参数指定的Annotation 将使用关键字作为Annotation 的别名。匿名的参数的别名将基于聚合函数的名称和模型的字段生成。只有引用单个字段的聚合表达式才可以使用匿名参数。其它所有形式都必须用关键字参数。

例如，如果你正在操作一个Blog 列表，你可能想知道每个Blog 有多少Entry：

```
>>> from django.db.models import Count
>>> q = Blog.objects.annotate(Count('entry'))
# The name of the first blog
>>> q[0].name
'Blogasaurus'
# The number of entries on the first blog
>>> q[0].entry__count
42
```

`Blog` 模型本身没有定义 `entry__count` 属性，但是通过使用一个关键字参数来指定聚合函数，你可以控制Annotation 的名称：

```
>>> q = Blog.objects.annotate(number_of_entries=Count('entry'))
# The number of entries on the first blog, using the name provid
ed
>>> q[0].number_of_entries
42
```

聚合的深入讨论，参见 聚合主题的指南。

order_by

`order_by` (**fields*)

默认情况下， `QuerySet` 根据模型 `Meta` 类的 `ordering` 选项排序。你可以使用 `order_by` 方法给每个 `QuerySet` 指定特定的排序。

例如：

```
Entry.objects.filter(pub_date__year=2005).order_by('-pub_date',
'headline')
```

上面的结果将按照 `pub_date` 降序排序，然后再按照 `headline` 升序排序。 `"-pub_date"` 前面的负号表示降序排序。隐式的是升序排序。若要随机排序，请使用 `"?"` ，像这样：

```
Entry.objects.order_by('?')
```

注： `order_by('?')` 查询可能耗费资源且很慢，这取决于使用的数据库。

若要按照另外一个模型中的字段排序，可以使用查询关联模型时的语法。即通过字段的名称后面跟上两个下划线（ `__` ），再跟上新模型中的字段的名称，直至你希望连接的模型。例如：

```
Entry.objects.order_by('blog__name', 'headline')
```

如果排序的字段与另外一个模型关联，Django 将使用关联的模型的默认排序，或者如果没有指定 `Meta.ordering` 将通过关联的模型的主键排序。 例如，因为 `Blog` 模型没有指定默认的排序：

```
Entry.objects.order_by('blog')
```

... 等同于：

```
Entry.objects.order_by('blog__id')
```

如果 `Blog` 设置 `ordering = ['name']` ，那么第一个QuerySet 将等同于：

```
Entry.objects.order_by('blog__name')
```

通过关联字段排序QuerySet 还能够不用带来JOIN 产生的花费，方法是引用关联字段的 `_id` ：

```
# No Join
Entry.objects.order_by('blog_id')

# Join
Entry.objects.order_by('blog__id')
```

New in Django 1.7:

QuerySet 通过关联字段进行排序不用带来JOIN 产生的开销。

你还可以通过调用表达式的 `asc()` 或者 `desc()` ，根据查询表达式排序：

```
Entry.objects.order_by(Coalesce('summary', 'headline').desc())
```

New in Django 1.8:

增加根据查询表达式排序。

如果你还用到 `distinct()` ，在根据关联模型中的字段排序时要小心。 `distinct()` 中有一个备注讲述关联模型的排序如何对结果产生影响。

注

指定一个多值字段来排序结果（例如，一个 `ManyToManyField` 字段或者 `ForeignKey` 字段的反向关联）。

考虑下面的情况：

```
class Event(Model):
    parent = models.ForeignKey('self', related_name='children')
    date = models.DateField()

Event.objects.order_by('children__date')
```

这里，每个 `Event` 可能有多个潜在的排序数据；each `Event` with multiple `children` will be returned multiple times into the new `QuerySet` that `order_by()` creates. 换句话说, 用 `order_by()` 方法对 `QuerySet` 对象进行操作会返回一个扩大版的新QuerySet对象——新增的条目也许并没有什么卵用，你也用不着它们。

因此，当你使用多值字段对结果进行排序时要格外小心。如果，您可以确保每个订单项只有一个订购数据，这种方法不会出现问题。如果不确定，请确保结果是你期望的。

没有方法指定排序是否考虑大小写。对于大小写的敏感性，Django 将根据数据库中的排序方式排序结果。

你可以通过 `Lower` 将一个字段转换为小写来排序，它将达到大小写一致的排序：

```
Entry.objects.order_by(Lower('headline').desc())
```

New in Django 1.8:

新增根据表达式如 `Lower` 来排序。

如果你不想对查询做任何排序，即使是默认的排序，可以不带参数调用 `order_by()` 。

你可以通过检查 `QuerySet.ordered` 属性来知道查询是否是排序的，如果 `QuerySet` 有任何方式的排序它将为 `True` 。

每个 `order_by()` 都将清除前面的任何排序。例如，下面的查询将按照 `pub_date` 排序，而不是 `headline` ：

```
Entry.objects.order_by('headline').order_by('pub_date')
```

警告

排序不是没有开销的操作。添加到排序中的每个字段都将带来数据库的开销。添加的每个外键也都将隐式包含进它的默认排序。

reverse

`reverse` ()

`reverse()` 方法反向排序QuerySet 中返回的元素。第二次调用 `reverse()` 将恢复到原有的排序。

如要获取QuerySet 中最后五个元素，你可以这样做：

```
my_queryset.reverse()[:5]
```

注意，这与Python 中从一个序列的末尾进行切片有点不一样。上面的例子将首先返回最后一个元素，然后是倒数第二个元素，以此类推。如果我们有一个Python 序列，当我们查看 `seq[-5:]` 时，我们将一下子得到倒数五个元素。Django 不支持这种访问模型（从末尾进行切片），因为它不可能利用SQL 高效地实现。

同时还要注意， `reverse()` 应该只在一个已经定义排序的 `QuerySet` 上调用（例如，在一个定义了默认排序的模型上，或者使用 `order_by()` 的时候）。如果 `QuerySet` 没有定义排序，调用 `reverse()` 将不会有任何效果（在调用 `reverse()` 之前没有定义排序，那么调用之后仍保持没有定义）。

distinct

`distinct` ([*fields])

返回一个在SQL 查询中使用 `SELECT DISTINCT` 的新 `QuerySet` 。它将去除查询结果中重复的行。

默认情况下， `QuerySet` 不会去除重复的行。在实际应用中，这一般不是个问题，因为像 `Blog.objects.all()` 这样的简单查询不会引入重复的行。但是，如果查询跨越多张表，当对 `QuerySet` 求值时就可能得到重复的结果。这时候你应该使用 `distinct()` 。

注

`order_by()` 调用中的任何字段都将包含在SQL 的 `SELECT` 列中。与 `distinct()` 一起使用时可能导致预计不到的结果。如果你根据关联模型的字段排序，这些fields将添加到查询的字段中，它们可能产生本应该是唯一的重复的行。因为多余的列没有出现在返回的结果中（它们只是为了支持排序），有时候看上去像是返回了不明确的结果。

类似地，如果您使用 `values()` 查询来限制所选择的列，则仍然会涉及任何 `order_by()` （或默认模型排序）影响结果的唯一性。

这里的道德是，如果你使用 `distinct()` 小心有关的模型排序。类似地，当一起使用 `distinct()` 和 `values()` 时，请注意字段在不在 `values()`

在PostgreSQL上，您可以传递位置参数（ `* fields` ），以便指定 `DISTINCT` 应该应用的字段的名称。这转换为 `SELECT DISTINCT ON` SQL查询。这里有区别。对于正常的 `distinct()` 调用，数据库在确定哪些行不同时比较每行中的每个字段。对于具有指定字段名称的 `distinct()` 调用，数据库将仅比较指定的字段名称。

注意

当你指定字段名称时，必须在 `QuerySet` 中提供 `order_by()` ，而且 `order_by()` 中的字段必须以 `distinct()` 中的字段相同开始并且顺序相同。

例如， `SELECT DISTINCT ON （a）` 列 `a` 中的每个值。如果你没有指定一个顺序，你会得到一个任意的行。

示例（除第一个示例外，其他示例都只能在PostgreSQL 上工作）：

```
>>> Author.objects.distinct()
[...]

>>> Entry.objects.order_by('pub_date').distinct('pub_date')
[...]

>>> Entry.objects.order_by('blog').distinct('blog')
[...]

>>> Entry.objects.order_by('author', 'pub_date').distinct('autho
r', 'pub_date')
[...]

>>> Entry.objects.order_by('blog__name', 'mod_date').distinct('b
log__name', 'mod_date')
[...]

>>> Entry.objects.order_by('author', 'pub_date').distinct('autho
r')
[...]
```

注释

请记住，`order_by()` 使用已定义的任何默认相关模型排序。您可能需要通过关系 `_id` 或引用字段显式排序，以确保 `DISTINCT ON` 在 `ORDER BY` 子句的开头。例如，如果 `Blog` 模型通过 `name` 定义 排序 ：

```
Entry.objects.order_by('blog').distinct('blog')
```

...无法工作，因为查询将按 `blog__name` 排序，从而使 `DISTINCT ON` 表达式不匹配。你必须按照关系<cite>_id</cite>字段（在这种情况下为 `blog_id` ）或引用的（ `blog__pk` ）显式排序来确保两个表达式都匹配。

values

`values` (*fields*)

返回一个 `ValuesQuerySet` —— `QuerySet` 的一个子类，迭代时返回字典而不是模型实例对象。

每个字典表示一个对象，键对应于模型对象的属性名称。

下面的例子将 `values()` 与普通的模型对象进行比较：

```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
[<Blog: Beatles Blog>]

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Be
atles news.'}]
```

`values()` 接收可选的位置参数 `*fields` ，它指定 `SELECT` 应该限制哪些字段。如果指定字段，每个字典将只包含指定的字段的键/值。如果没有指定字段，每个字典将包含数据库表中所有字段的键和值。

例如：

```
>>> Blog.objects.values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Be
atles news.'}],
>>> Blog.objects.values('id', 'name')
[{'id': 1, 'name': 'Beatles Blog'}]
```

值得注意的几点：

- 如果你有一个字段 `foo` 是一个 `ForeignKey`，默认的 `values()` 调用返回的字典将有一个叫做 `foo_id` 的键，因为这是保存实际的值的那个隐藏的模型属性的名称（ `foo` 属性引用关联的模型）。当你调用 `values()` 并传递字段的名称，传递 `foo` 或 `foo_id` 都可以，得到的结果是相同的（字典的键会与你传递的字段名匹配）。

  例如：

  ```
  &gt;&gt;&gt; Entry.objects.values()
  [{'blog_id': 1, 'headline': 'First Entry', ...}, ...]

  &gt;&gt;&gt; Entry.objects.values('blog')
  [{'blog': 1}, ...]

  &gt;&gt;&gt; Entry.objects.values('blog_id')
  [{'blog_id': 1}, ...]
  ```

- 当 `values()` 与 `distinct()` 一起使用时，注意排序可能影响最终的结果。详细信息参见 `distinct()` 中的备注。
- 如果 `values()` 子句位于 `extra()` 调用之后， `extra()` 中的 `select` 参数定义的字段必须显式包含在 `values()` 调用中。 `values()` 调用后面的 `extra()` 调用将忽略选择的额外的字段。
- 在 `values()` 之后调用 `only()` 和 `defer()` 不太合理，所以将引发一个 `NotImplementedError` 。

New in Django 1.7:

新增最后一点。以前，在 `values()` 之后调用 `only()` 和 `defer()` 是允许的，但是它要么会崩溃要么返回错误的结果。

`ValuesQuerySet` 用于你知道你只需要字段的一小部分，而不需要用到模型实例对象的函数。只选择用到的字段当然更高效。

最后，要注意 `ValuesQuerySet` 是 `QuerySet` 的子类，它实现了大部分相同的方法。你可以对它调用 `filter()` 、 `order_by()` 等等。这表示下面的两个调用完全相同：

```
Blog.objects.values().order_by('id')
Blog.objects.order_by('id').values()
```

Django 的作者喜欢将影响SQL 的方法放在前面，然后放置影响输出的方法（例如 `values()` ），但是实际上无所谓。这是卖弄你个性的好机会。

你可以通过 `OneToOneField` 、 `ForeignKey` 和 `ManyToManyField` 属性反向引用关联的模型的字段：

```
Blog.objects.values('name', 'entry__headline')
[{'name': 'My blog', 'entry__headline': 'An entry'},
     {'name': 'My blog', 'entry__headline': 'Another entry'}, ..
.]
```

警告

因为 `ManyToManyField` 字段和反向关联可能有多个关联的行，包含它们可能导致结果集的倍数放大。如果你在 `values()` 查询中包含多个这样的字段将更加明显，这种情况下将返回所有可能的组合。

values_list

`values_list` (**fields*, *flat=False*)

与 `values()` 类似，只是在迭代时返回的是元组而不是字典。每个元组包含传递给 `values_list()` 调用的字段的值 —— 所以第一个元素为第一个字段，以此类推。例如：

```
>>> Entry.objects.values_list('id', 'headline')
[(1, 'First entry'), ...]
```

如果只传递一个字段，你还可以传递 `flat` 参数。如果为 `True` ，它表示返回的结果为单个值而不是元组。一个例子会让它们的区别更加清晰：

```
>>> Entry.objects.values_list('id').order_by('id')
[(1,), (2,), (3,), ...]

>>> Entry.objects.values_list('id', flat=True).order_by('id')
[1, 2, 3, ...]
```

如果有多个字段，传递 `flat` 将发生错误。

如果你不传递任何值给 `values_list()` ，它将返回模型中的所有字段，以它们在模型中定义的顺序。

注意，这个方法返回 `ValuesListQuerySet` 。这个类的行为类似列表。大部分时候它足够用了，但是如果你需要一个真实的Python 列表对象，可以对它调用 `list()` ，这将会对查询集求值。

dates

`dates` (*field*, *kind*, *order='ASC'*)

返回 `DateQuerySet` - `QuerySet` ，其计算结果为 `datetime.date` 对象列表，表示特定种类的所有可用日期 `QuerySet` 。

`field` 应为模型的 `DateField` 的名称。 `kind` 应为 `"year"` 、 `"month"` 或 `"day"` 。隐式的是升序排序。若要随机排序，请使用 `"?"` ，像这样：

- `"year"` 返回对应该field的所有不同年份值的list。
- `“month”` 返回字段的所有不同年/月值的列表。
- `“day”` 返回字段的所有不同年/月/日值的列表。

`order` （默认为 `“ASC”` ）应为 `'ASC'` 或 `'DESC'` 。它t指定如何排序结果。

例子：

```
>>> Entry.objects.dates('pub_date', 'year')
[datetime.date(2005, 1, 1)]
>>> Entry.objects.dates('pub_date', 'month')
[datetime.date(2005, 2, 1), datetime.date(2005, 3, 1)]
>>> Entry.objects.dates('pub_date', 'day')
[datetime.date(2005, 2, 20), datetime.date(2005, 3, 20)]
>>> Entry.objects.dates('pub_date', 'day', order='DESC')
[datetime.date(2005, 3, 20), datetime.date(2005, 2, 20)]
>>> Entry.objects.filter(headline__contains='Lennon').dates('pub
_date', 'day')
[datetime.date(2005, 3, 20)]
```

datetimes

`datetimes` (*field_name*, *kind*, *order='ASC'*, *tzinfo=None*)

返回 `QuerySet` ，其计算为 `datetime.datetime` 对象的列表，表示 `QuerySet` 内容中特定种类的所有可用日期。

`field_name` 应为模型的 `DateTimeField` 的名称。

`种` 应为 `“year”` ， `“month”` ， `“day”` ， `“hour”` ， `“分钟”` 或 `“秒”` 。结果列表中的每个 `datetime.datetime` 对象被“截断”到给定的 `类型` 。

`order` , 默认为 `'ASC'` , 可选项为 `'ASC'` 或者 `'DESC'` . 这个选项指定了返回结果的排序方式。

`tzinfo` 定义在截断之前将数据时间转换到的时区。实际上，给定的datetime具有不同的表示，这取决于使用的时区。此参数必须是 `datetime.tzinfo` 对象。如果它 `无` ，Django使用当前时区。当 `USE_TZ` 为 `False` 时，它不起作用。

注意

此函数直接在数据库中执行时区转换。因此，您的数据库必须能够解释 `tzinfo.tzname(None)` 的值。这转化为以下要求：

- SQLite：install pytz - 转换实际上是在Python中执行的。
- PostgreSQL：没有要求（见时区）。
- Oracle：无要求（请参阅选择时区文件）。
- MySQL：安装pytz，并使用mysql_tzinfo_to_sql加载时区表。

none

`none` ()

调用none()将创建一个从不返回任何对象的查询集，并且在访问结果时不会执行任何查询。qs.none()查询集是 `EmptyQuerySet` 的一个实例。

例子：

```
>>> Entry.objects.none()
[]
>>> from django.db.models.query import EmptyQuerySet
>>> isinstance(Entry.objects.none(), EmptyQuerySet)
True
```

all

`all` ()

返回当前 `QuerySet` （或 `QuerySet` 子类） 的副本。它可以用于在你希望传递一个模型管理器或 `QuerySet` 并对结果做进一步过滤的情况。不管对哪一种对象调用 `all()` ，你都将获得一个可以工作的 `QuerySet` 。

当对 `QuerySet` 进行求值时，它通常会缓存其结果。如果数据库中的数据在 `QuerySet` 求值之后可能已经改变，你可以通过在以前求值过的 `QuerySet` 上调用相同的 `all()` 查询以获得更新后的结果。

select_related

`select_related` (**fields*)

返回一个 `QuerySet` ，当执行它的查询时它沿着外键关系查询关联的对象的数据。它会生成一个复杂的查询并引起性能的损耗，但是在以后使用外键关系时将不需要数据库查询。

下面的例子解释了普通查询和 `select_related()` 查询的区别。下面是一个标准的查询：

```
# Hits the database.
e = Entry.objects.get(id=5)

# Hits the database again to get the related Blog object.
b = e.blog
```

下面是一个 `select_related` 查询：

```
# Hits the database.
e = Entry.objects.select_related('blog').get(id=5)

# Doesn't hit the database, because e.blog has been prepopulated
# in the previous query.
b = e.blog
```

`select_related()` 可用于任何对象的查询集：

```
from django.utils import timezone

# Find all the blogs with entries scheduled to be published in t
he future.
blogs = set()

for e in Entry.objects.filter(pub_date__gt=timezone.now()).selec
t_related('blog'):
    # Without select_related(), this would make a database query
 for each
    # loop iteration in order to fetch the related blog for each
 entry.
    blogs.add(e.blog)
```

`filter()` 和 `select_related()` 链的顺序不重要。下面的查询集是等同的：

```
Entry.objects.filter(pub_date__gt=timezone.now()).select_related
('blog')
Entry.objects.select_related('blog').filter(pub_date__gt=timezon
e.now())
```

你可以沿着外键查询。如果你有以下模型：

```
from django.db import models

class City(models.Model):
    # ...
    pass

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(City)

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person)
```

... 那么 `Book.objects.select_related('author__hometown').get(id=4)` 调用将缓存关联的 `Person` 和关联的 `City` ：

```
b = Book.objects.select_related('author__hometown').get(id=4)
p = b.author         # Doesn't hit the database.
c = p.hometown       # Doesn't hit the database.

b = Book.objects.get(id=4) # No select_related() in this example
.
p = b.author         # Hits the database.
c = p.hometown       # Hits the database.
```

在传递给 `select_related()` 的字段中，你可以使用任何 `ForeignKey` 和 `OneToOneField` 。

在传递给 `select_related` 的字段中，你还可以反向引用 `OneToOneField` —— 也就是说，你可以回溯到定义 `OneToOneField` 的字段。此时，可以使用关联对象字段的 `related_name` ，而不要指定字段的名称。

有些情况下，你希望对很多对象调用 `select_related()` ，或者你不知道所有的关联关系。在这些情况下，可以调用不带参数的 `select_related()` 。它将查找能找到的所有不可为空外键 —— 可以为空的外键必须明确指定。大部分情况下不建议这样做，因为它会使得底层的查询非常复杂并且返回的很多数据都不是真实需要的。

如果你需要清除 `QuerySet` 上以前的 `select_related` 添加的关联字段，可以传递一个 `None` 作为参数：

```
>>> without_relations = queryset.select_related(None)
```

链式调用 `select_related` 的工作方式与其它方法类似 —— 也就是说，`select_related('foo', 'bar')` 等同于 `select_related('foo').select_related('bar')`。

Changed in Django 1.7:

在以前，后者等同于 `select_related('bar')`。

prefetch_related

`prefetch_related` (**lookups*)

返回 `QuerySet`，它将在单个批处理中自动检索每个指定查找的相关对象。

这具有与 `select_related` 类似的目的，两者都被设计为阻止由访问相关对象而导致的数据库查询的泛滥，但是策略是完全不同的。

`select_related` 通过创建SQL连接并在 `SELECT` 语句中包括相关对象的字段来工作。因此，`select_related` 在同一数据库查询中获取相关对象。然而，为了避免由于跨越"多个"关系而导致的大得多的结果集，`select_related` 限于单值关系 - 外键和一对一关系。

`prefetch_related`，另一方面，为每个关系单独查找，并在Python中"加入"。这允许它预取多对多和多对一对象，除了外键和一对一关系，它们不能使用 `select_related` 来完成。`select_related`。它还支持 `GenericRelation` 和 `GenericForeignKey` 的预取。

例如，假设您有这些模型：

```
from django.db import models

class Topping(models.Model):
    name = models.CharField(max_length=30)

class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)

    def __str__(self):              # __unicode__ on Python 2
        return "%s (%s)" % (self.name, ", ".join(topping.name
                                                  for topping in
self.toppings.all()))
```

并运行：

```
>>> Pizza.objects.all()
["Hawaiian (ham, pineapple)", "Seafood (prawns, smoked salmon)".
..
```

问题是每次 `Pizza .__ str __()` 要求 `self.toppings.all()` 它必须查询数据库，因此 `Pizza.objects .all()` 将在Pizza `QuerySet` 中的每个项目的Toppings表上运行查询。

我们可以使用 `prefetch_related` 减少为只有两个查询：

```
>>> Pizza.objects.all().prefetch_related('toppings')
```

这意味着检索到的每个 `Pizza` 都会执行 `self.toppings.all()` ；现在每次调用 `self.toppings.all()` ，而不是去数据库的项目，它会在预取的 `QuerySet` 缓存中找到它们填充在单个查询中。

也就是说，所有相关的配料将在单个查询中提取，并用于使具有相关结果的预填充缓存的 `QuerySets` ；这些 `QuerySets` 然后在 `self.toppings.all()` 调用中使用。

`prefetch_related()` 中的附加查询在 `QuerySet` 开始计算并且主查询已执行后执行。

请注意，主要 `QuerySet` 的结果缓存和所有指定的相关对象将被完全加载到内存中。这改变了 `QuerySets` 的典型行为，通常尽量避免在需要之前将所有对象加载到内存中，即使在数据库中执行了查询之后。

注意

请记住，与 `QuerySets` 一样，任何后续的链接方法隐含不同的数据库查询将忽略以前缓存的结果，并使用新的数据库查询检索数据。所以，如果你写下面的话：

```
>>> pizzas = Pizza.objects.prefetch_related('toppings')
>>> [list(pizza.toppings.filter(spicy=True)) for pizza in pizzas
]
```

...然后事实，已经预取的 `pizza.toppings.all()` 不会帮助你。 `prefetch_related（'toppings'）` 隐含 `pizza.toppings.all()` ，但 `pizza.toppings.filter()` 是一个不同的查询。预取的缓存在这里不能帮助；实际上它伤害性能，因为你做了一个你没有使用的数据库查询。所以使用这个功能小心！

您还可以使用正常连接语法来执行相关字段的相关字段。假设我们有一个额外的模型上面的例子：

```
class Restaurant(models.Model):
    pizzas = models.ManyToMany(Pizza, related_name='restaurants'
)
    best_pizza = models.ForeignKey(Pizza, related_name='champion
ed_by')
```

以下都是合法的：

```
>>> Restaurant.objects.prefetch_related('pizzas__toppings')
```

这将预取所有比萨饼属于餐厅，所有浇头属于那些比萨饼。这将导致总共3个数据库查询 - 一个用于餐馆，一个用于比萨饼，一个用于浇头。

```
>>> Restaurant.objects.prefetch_related('best_pizza__toppings')
```

这将获取最好的比萨饼和每个餐厅最好的披萨的所有浇头。这将在3个数据库查询 - 一个为餐厅，一个为“最佳比萨饼”，一个为一个为浇头。

当然，也可以使用 `select_related` 来获取 `best_pizza` 关系，以将查询计数减少为2：

```
>>> Restaurant.objects.select_related('best_pizza').prefetch_rel
ated('best_pizza__toppings')
```

由于预取在主查询（其包括 `select_related` 所需的连接）之后执行，因此它能够检测到 `best_pizza` 对象已经被提取，并且请跳过重新获取它们。

链接 `prefetch_related` 调用将累积预取的查找。要清除任何 `prefetch_related` 行为，请传递 `None` 作为参数：

```
>>> non_prefetched = qs.prefetch_related(None)
```

使用 `prefetch_related` 时需要注意的一点是，查询创建的对象可以在它们相关的不同对象之间共享，即单个Python模型实例可以出现在树中的多个点返回的对象。这通常会与外键关系发生。通常这种行为不会是一个问题，并且实际上会节省内存和CPU时间。

虽然 `prefetch_related` 支持预取 `GenericForeignKey` 关系，但查询的数量将取决于数据。由于 `GenericForeignKey` 可以引用多个表中的数据，因此需要对每个引用的表进行一次查询，而不是对所有项进行一次查询。如果尚未提取相关行，则可能会对 `ContentType` 表执行其他查询。

`prefetch_related` 在大多数情况下将使用使用“IN”运算符的SQL查询来实现。这意味着对于一个大的 `QuerySet` ，可能会生成一个大的“IN”子句，根据数据库，在解析或执行SQL查询时可能会有性能问题。始终为您的使用情况配置文件！

请注意，如果您使用 `iterator()` 来运行查询，则会忽略 `prefetch_related()` 调用，因为这两个优化并没有意义。

New in Django 1.7.

您可以使用 `Prefetch` 对象进一步控制预取操作。

在其最简单的形式中， `Prefetch` 等效于传统的基于字符串的查找：

```
>>> Restaurant.objects.prefetch_related(Prefetch('pizzas__toppin
gs'))
```

您可以使用可选的 `queryset` 参数提供自定义查询集。这可以用于更改查询集的默认顺序：

```
>>> Restaurant.objects.prefetch_related(
...     Prefetch('pizzas__toppings', queryset=Toppings.objects.o
rder_by('name')))
```

或者在适当时调用 `select_related()` 以进一步减少查询数量：

```
>>> Pizza.objects.prefetch_related(
...     Prefetch('restaurants', queryset=Restaurant.objects.sele
ct_related('best_pizza')))
```

您还可以使用可选的 `to_attr` 参数将预取结果分配给自定义属性。结果将直接存储在列表中。

这允许使用不同的 `QuerySet` 预取相同的关系多次；例如：

```
>>> vegetarian_pizzas = Pizza.objects.filter(vegetarian=True)
>>> Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', to_attr='menu'),
...     Prefetch('pizzas', queryset=vegetarian_pizzas, to_attr='
vegetarian_menu'))
```

使用自定义 `to_attr` 创建的查找仍然可以像往常一样被其他查找遍历：

```
>>> vegetarian_pizzas = Pizza.objects.filter(vegetarian=True)
>>> Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', queryset=vegetarian_pizzas, to_attr='
vegetarian_menu'),
...     'vegetarian_menu__toppings')
```

在过滤预取结果时，建议使用 `to_attr` ，因为它比在相关管理器的缓存中存储过滤的结果更不明确：

```
>>> queryset = Pizza.objects.filter(vegetarian=True)
>>>
>>> # Recommended:
>>> restaurants = Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', queryset=queryset, to_attr='vegetaria
n_pizzas'))
>>> vegetarian_pizzas = restaurants[0].vegetarian_pizzas
>>>
>>> # Not recommended:
>>> restaurants = Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', queryset=queryset))
>>> vegetarian_pizzas = restaurants[0].pizzas.all()
```

自定义预取也适用于单个相关关系，如前 `ForeignKey` 或 `OneToOneField` 。一般来说，您希望对这些关系使用 `select_related()` ，但有很多情况下使用自定义 `QuerySet` 进行预取是有用的：

- 您想要使用在相关模型上执行进一步预取的 `QuerySet` 。
- 您希望仅预取相关对象的子集。
- You want to use performance optimization techniques like `deferred fields` :

```
&gt;&gt;&gt; queryset = Pizza.objects.only('name')
&gt;&gt;&gt;
&gt;&gt;&gt; restaurants = Restaurant.objects.prefetch_relat
ed(
...     Prefetch('best_pizza', queryset=queryset))
```

注意

查找的顺序很重要。

请看下面的例子：

```
>>> prefetch_related('pizzas__toppings', 'pizzas')
```

即使它是无序的，因为 `'pizzas__toppings'` 已经包含所有需要的信息，因此第二个参数 `'pizzas'` 实际上是多余的。

```
>>> prefetch_related('pizzas__toppings', Prefetch('pizzas', quer
yset=Pizza.objects.all()))
```

这将引发 `ValueError` ，因为尝试重新定义先前查看的查询的查询集。请注意，创建了隐式查询集，以作为 `'pizzas__toppings'` 查找的一部分遍历 `'pizzas'` 。

```
>>> prefetch_related('pizza_list__toppings', Prefetch('pizzas',
to_attr='pizza_list'))
```

这会触发 `AttributeError` ，因为 `'pizza_list'` 在处理 `'pizza_list__toppings'` 时不存在。

这种考虑不限于使用 `Prefetch` 对象。一些高级技术可能需要以特定顺序执行查找以避免创建额外的查询；因此建议始终仔细订购 `prefetch_related` 参数。

extra

`extra` (*select=None*, *where=None*, *params=None*, *tables=None*, *order_by=None*, *select_params=None*)

有些情况下，Django的查询语法难以简单的表达复杂的 `WHERE` 子句，对于这种情况, Django 提供了 `extra()` `QuerySet` 修改机制 — 它能在 `QuerySet` 生成的SQL从句中注入新子句

警告

无论何时你都需要非常小心的使用 `extra()` . 每次使用它时，您都应该转义用户可以使用 `params` 控制的任何参数，以防止SQL注入攻击。请详细了解*SQL injection protection*。

由于产品差异的原因，这些自定义的查询难以保障在不同的数据库之间兼容(因为你手写 SQL 代码的原因)，而且违背了 DRY 原则，所以如非必要，还是尽量避免写 extra。

extra可以指定一个或多个 `参数` ,例如 `select` , `where` or `tables` . 这些参数都不是必须的，但是你至少要使用一个

- `select`

  The `select` 参数可以让你在 `SELECT` 从句中添加其他字段信息，它应该是一个字典，存放着属性名到 SQL 从句的映射。

  例：

  ```
  Entry.objects.extra(select={'is_recent': "pub_date &gt; '200
  6-01-01'"})
  ```

  结果集中每个 `Entry` 对象都有一个额外的属性 `is_recent` , 它是一个布尔值，表示 Entry对象的 `pub_date` 是否晚于 Jan. 1, 2006.

Django 会直接在 `SELECT` 中加入对应的 SQL 片断，所以转换后的 SQL 如下：

```
SELECT blog_entry.*, (pub_date &gt; '2006-01-01') AS is_rece
nt
FROM blog_entry;
```

下面这个例子更复杂一些；它会在每个 `Blog` 对象中添加一个 `entry_count` 属性，它会运行一个子查询，得到相关联的 `Entry` 对象的数量：

```
Blog.objects.extra(
    select={
        'entry_count': 'SELECT COUNT(*) FROM blog_entry WHER
E blog_entry.blog_id = blog_blog.id'
    },
)
```

在上面这个特例中，我们要了解这个事实，就是 `blog_blog` 表已经存在于 `FROM` 从句中

上面例子的结果SQL将是：

```
SELECT blog_blog.*, (SELECT COUNT(*) FROM blog_entry WHERE b
log_entry.blog_id = blog_blog.id) AS entry_count
FROM blog_blog;
```

要注意的是，大多数数据库需要在子句两端添加括号，而在 Django 的 `select` 从句中却无须这样。 另请注意，某些数据库后端（如某些MySQL版本）不支持子查询。

在少数情况下，您可能希望将参数传递到 `extra（select = ...）` 中的SQL片段。为此，请使用 `select_params` 参数。由于 `select_params` 是一个序列，并且 `select` 属性是字典，因此需要注意，以便参数与额外的选择片段正确匹配。在这种情况下，您应该使用 `collections.` OrderedDict用于 `select` 值，而不仅仅是一个普通的Python字典。

这将工作，例如：

```
Blog.objects.extra(
    select=OrderedDict([('a', '%s'), ('b', '%s')]),
    select_params=('one', 'two'))
```

如果您需要在选择字符串中使用文本 `%s` ，请使用序列 `%%s` 。

Changed in Django 1.8:

在1.8之前，您无法逃离文本 `%s` 。

- `where` / `tables`

  您可以使用 `其中` 定义显式SQL `WHERE` 子句 - 也许执行非显式连接。您可以使用 `表` 手动将表添加到SQL `FROM` 子句。

  `其中` 和 `表` 都接受字符串列表。所有 `其中` 参数均为“与”任何其他搜索条件。

  例：

  ```
  Entry.objects.extra(where=["foo='a' OR bar = 'a'", "baz = 'a'"])
  ```

  ...翻译（大致）以下SQL：

  ```
  SELECT * FROM blog_entry WHERE (foo='a' OR bar='a') AND (baz='a')
  ```

  如果您要指定已在查询中使用的表，请在使用 `tables` 参数时小心。当您通过 `tables` 参数添加额外的表时，Django假定您希望该表包含额外的时间（如果已包括）。这会产生一个问题，因为表名将会被赋予一个别名。如果表在SQL语句中多次出现，则第二次和后续出现必须使用别名，以便数据库可以区分它们。如果您指的是在额外的 `where` 参数中添加的额外表，这将导致错误。

  通常，您只需添加尚未显示在查询中的额外表。然而，如果发生上述情况，则有几种解决方案。首先，看看你是否可以不包括额外的表，并使用已经在查询中的一个。如果不可能，请将 `extra()` 调用放在查询集结构的前面，以便您的表是该表的第一次使用。最后，如果所有其他失败，请查看生成的查询并重写 `where` 添加以使用给您的额外表的别名。每次以相同的方式构造查询集时，别名将是相同的，因此您可以依靠别名不更改。

- `order_by`

  如果您需要使用通过 `extra()` 包含的一些新字段或表来对结果查询进行排序，请使用 `order_by` 参数 `extra()` 这些字符串应该是模型字段（如查询集上的正常 `order_by()` 方法），形式为 `table_name.column_name` 或您在 `select` 参数到 `extra()` 。

  例如：

  ```
  q = Entry.objects.extra(select={'is_recent': "pub_date &gt; '2006-01-01'"})
  q = q.extra(order_by = ['-is_recent'])
  ```

这会将 `is_recent` 的所有项目排序到结果集的前面（ `True` 在 `False` 之前按降序排序）。

顺便说一句，你可以对 `extra()` 进行多次调用，它会按照你的期望（每次添加新的约束）运行。

- `params`

  上述 `where` 参数可以使用标准Python数据库字符串占位符 - `'%s'` 来指示数据库引擎应自动引用的参数。 `params` 参数是要替换的任何额外参数的列表。

  例：

  ```
  Entry.objects.extra(where=['headline=%s'], params=['Lennon']
  )
  ```

  始终使用 `params` 而不是将值直接嵌入 `where` ，因为 `params` 会确保根据您的特定后端正确引用值。例如，引号将被正确转义。

  坏：

  ```
  Entry.objects.extra(where=["headline='Lennon'"])
  ```

  好：

  ```
  Entry.objects.extra(where=['headline=%s'], params=['Lennon']
  )
  ```

警告

如果您正在对MySQL执行查询，请注意，MySQL的静默类型强制可能会在混合类型时导致意外的结果。If you query on a string type column, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison.例如，如果表包含值 `'abc'` ， `'def'` ，并查询 `WHERE mycolumn = 0` ，两行都将匹配。为了防止这种情况，请在使用查询中的值之前执行正确的类型转换。

defer

`defer` (**fields*)

在一些复杂的数据建模情况下，您的模型可能包含大量字段，其中一些可能包含大量数据（例如，文本字段），或者需要昂贵的处理来将它们转换为Python对象。如果您在某些情况下使用查询集的结果，当您最初获取数据时不知道是否需要这些特定字段，可以告诉Django不要从数据库中检索它们。

这是通过传递字段名称不加载到 `defer()` ：

```
Entry.objects.defer("headline", "body")
```

具有延迟字段的查询集仍将返回模型实例。如果您访问该字段（一次一个，而不是一次所有的延迟字段），将从数据库中检索每个延迟字段。

您可以多次调用 `defer()` 。每个调用都向延迟集添加新字段：

```
# Defers both the body and headline fields.
Entry.objects.defer("body").filter(rating=5).defer("headline")
```

字段添加到延迟集的顺序无关紧要。调用具有已延迟的字段名称的 `defer()` 是无害的（该字段仍将被延迟）。

您可以使用标准的双下划线符号来分隔相关字段，从而推迟相关模型中的字段加载（如果相关模型通过 `select_related()` 加载）

```
Blog.objects.select_related().defer("entry__headline", "entry__b
ody")
```

如果要清除延迟字段集，请将 `None` 作为参数传递到 `defer()` ：

```
# Load all fields immediately.
my_queryset.defer(None)
```

模型中的某些字段不会被延迟，即使您要求它们。你永远不能推迟加载主键。如果您使用 `select_related()` 检索相关模型，则不应推迟从主模型连接到相关模型的字段的加载，否则将导致错误。

注意

`defer()` 方法（及其表兄弟， `only()` ）仅适用于高级用例。They provide an optimization for when you have analyzed your queries closely and understand *exactly* what information you need and have measured that the difference between returning the fields you need and the full set of fields for the model will be significant.

即使你认为你是在高级用例的情况下，只使用**defer()**，当你不能，在查询集加载时，确定是否需要额外的字段或。如果您经常加载和使用特定的数据子集，最好的选择是规范化模型，并将未加载的数据放入单独的模型（和数据库表）。如果列必须由于某种原因保留在一个表中，请创建一个具有 Meta.managed = / t4> （请参阅 `managed attribute` 文档），只包含您通常需要加载和使用的字段否则调用 `defer()` 。这使得你的代码对读者更加明确，稍微更快一些，并且在Python进程中消耗更少的内存。

例如，这两个模型使用相同的底层数据库表：

```
class CommonlyUsedModel(models.Model):
    f1 = models.CharField(max_length=10)

    class Meta:
        managed = False
        db_table = 'app_largetable'

class ManagedModel(models.Model):
    f1 = models.CharField(max_length=10)
    f2 = models.CharField(max_length=10)

    class Meta:
        db_table = 'app_largetable'

# Two equivalent QuerySets:
CommonlyUsedModel.objects.all()
ManagedModel.objects.all().defer('f2')
```

如果许多字段需要在非托管模型中复制，最好使用共享字段创建抽象模型，然后使非托管模型和托管模型从抽象模型继承。

注意

当对具有延迟字段的实例调用 `save()` 时，仅保存加载的字段。有关详细信息，请参见 `save()` 。

only

`only` (*fields*)

`only()` 方法或多或少与 `defer()` 相反。您调用它时，应该在检索模型时延迟的字段。如果你有一个模型几乎所有的字段需要延迟，使用 `only()` 指定补充的字段集可以导致更简单的代码。

假设您有一个包含字段 `名称` ， `年龄` 和 `传记` 的模型。以下两个查询集是相同的，就延迟字段而言：

```
Person.objects.defer("age", "biography")
Person.objects.only("name")
```

每当您调用 `（）` 时，取代立即加载的字段集。方法的名称是助记符：只有这些字段立即加载；其余的都被推迟。因此，对 `only()` 的连续调用仅导致所考虑的最后字段：

```
# This will defer all fields except the headline.
Entry.objects.only("body", "rating").only("headline")
```

由于 `defer()` 以递增方式动作（向延迟列表中添加字段），因此您可以将调用结合到 `only()` 和 `defer()`

```
# Final result is that everything except "headline" is deferred.
Entry.objects.only("headline", "body").defer("body")

# Final result loads headline and body immediately (only() repla
ces any
# existing set of fields).
Entry.objects.defer("body").only("headline", "body")
```

`defer()` 文档注释中的所有注意事项也适用于 `only()` 。使用它谨慎，只有耗尽了你的其他选项。

使用 `only()` 并省略使用 `select_related()` 请求的字段也是错误。

注意

当对具有延迟字段的实例调用 `save()` 时，仅保存加载的字段。有关详细信息，请参见 `save()` 。

using

`using` (*alias*)

如果你使用多个数据库，这个方法用于控制 `QuerySet` 将在哪个数据库上求值。这个方法的唯一参数是数据库的别名，定义在 DATABASES 。

例如：

```
# queries the database with the 'default' alias.
>>> Entry.objects.all()

# queries the database with the 'backup' alias
>>> Entry.objects.using('backup')
```

select_for_update

`select_for_update` (*nowait=False*)

返回一个 queryset ，会锁定相关行直到事务结束。在支持的数据库上面产生一个 `SELECT ...` FORUPDATE语句

例如:

```
entries = Entry.objects.select_for_update().filter(author=reques
t.user)
```

所有匹配的行将被锁定，直到事务结束。这意味着可以通过锁防止数据被其它事务修改。

一般情况下如果其他事务锁定了相关行，那么本查询将被阻塞，直到锁被释放。如果这不是你想要的行为，请使用 `select_for_update(nowait=True)` . 这将使查询不阻塞。如果其它事务持有冲突的锁, 那么查询将引发 `DatabaseError` 异常.

目前 `postgresql_psycopg2` , `oracle` 和 `mysql` 数据库后端 `select_for_update()` . 但是 MySQL 不支持 `nowait` 参数。显然，用户应该检查后端的支持情况。

当在不支持 `nowait` 功能的数据库后端(例如 MySql) 使用 `nowait=True` 参数调用 `select_for_update()` 时将引发 `DatabaseError` 异常. 这是防止意外造成代码被阻塞。

在自动提交模式下使用 `select_for_update()` 将引发 `TransactionManagementError` 异常，原因是自动提交模式下不支持锁定行。如果允许这个调用，那么可能造成数据损坏，而且这个功能很容易在事务外被调用。

对于不支持 `SELECT ...` FORUPDATE的后端 (例如SQLite) select_for_update() 将没有效果。

Changed in Django 1.6.3:

在自动提交模式下，使用 `select_for_update()` 执行查询现在是一个错误。在1.6系列的早期版本中，它是一个无操作。

警告

Although `select_for_update()` normally fails in autocommit mode, since `TestCase` automatically wraps each test in a transaction, calling `select_for_update()` in a `TestCase` even outside an `atomic()` block will (perhaps unexpectedly) pass without raising a `TransactionManagementError` . 要正确测试 `select_for_update()` ，您应该使用 `TransactionTestCase` 。

raw

`raw` (*raw_query*, *params=None*, *translations=None*)

Changed in Django 1.7:

`raw` 移动到 `QuerySet` 类中。以前，它只位于 `Manager` 中。

接收一个原始的SQL 查询，执行它并返回一个 `django.db.models.query.RawQuerySet` 实例。这个 `RawQuerySet` 实例可以迭代以提供实例对象，就像普通的 `QuerySet` 一样。

更多信息参见执行原始的SQL 查询。

警告

`raw()` 永远触发一个新的查询，而与之前的filter 无关。因此，它通常应该从 `Manager` 或一个全新的 `QuerySet` 实例调用。

不会返回QuerySets的方法

以下 `QuerySet` 方法评估 `QuerySet` 并返回而不是 a `QuerySet` 。

这些方法不使用高速缓存（请参阅*Caching and QuerySets*）。这些方法每次被调用的时候都会查询数据库。

get

`get` (***kwargs*)

返回按照查询参数匹配到的对象，参数的格式应该符合 Field lookups的要求.

如果匹配到的对象个数不只一个的话， `get()` 将会触发 `MultipleObjectsReturned` 异常. `MultipleObjectsReturned` 异常是模型类的属性.

如果根据给出的参数匹配不到对象的话， `get()` 将触发 `DoesNotExist` 异常. 这个异常是模型类的属性. 例：

```
Entry.objects.get(id='foo') # raises Entry.DoesNotExist
```

`DoesNotExist` 异常从 `django.core.exceptions.ObjectDoesNotExist` 继承，因此您可以定位多个 `DoesNotExist` 异常。例：

```
from django.core.exceptions import ObjectDoesNotExist
try:
    e = Entry.objects.get(id=3)
    b = Blog.objects.get(id=1)
except ObjectDoesNotExist:
    print("Either the entry or blog doesn't exist.")
```

create

`create` (***kwargs*)

一个在一步操作中同时创建对象并且保存的便捷方法. 所以:

```
p = Person.objects.create(first_name="Bruce", last_name="Springs
teen")
```

和:

```
p = Person(first_name="Bruce", last_name="Springsteen")
p.save(force_insert=True)
```

是等同的.

参数 *force_insert* 在其他的文档中有介绍, 它意味着一个新的对象一定会被创建. 正常情况中，你不必要担心这点. 然而, 如果你的model中有一个你手动设置主键， 并且这个值已经存在于数据库中, 调用 `create()` 将会失败并且触发 `IntegrityError` 因为主键必须是唯一的. 如果你手动设置了主键，做好异常处理的准备.

get_or_create

`get_or_create` (*defaults=None*, ***kwargs*)

一个通过给出的 `kwargs` 来查询对象的便捷方法（如果你的模型中的所有字段都有默认值，可以为空），需要的话创建一个对象。

返回一个由 `(object, created)` 组成的元组，元组中的 `object` 是一个查询到的或者是被创建的对象， `created` 是一个表示是否创建了新的对象的布尔值。

这主要用作样板代码的一种快捷方式。例如：

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lenno
n')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday
=date(1940, 10, 9))
    obj.save()
```

如果模型的字段数量较大的话，这种模式就变的非常不易用了。上面的示例可以用 `get_or_create()` 重写：

```
obj, created = Person.objects.get_or_create(first_name='John', l
ast_name='Lennon',
                  defaults={'birthday': date(1940, 10, 9)})
```

任何传递给 `get_or_create()` 的关键字参数，除了一个可选的 `defaults` ，都将传递给 `get()` 调用。如果查找到一个对象， `get_or_create()` 返回一个包含匹配到的对象以及 `False` 组成的元组。如果查找到的对象超过一个以上， `get_or_create` 将引发 `MultipleObjectsReturned` 。如果查找不到对象， `get_or_create()` 将会实例化并保存一个新的对象，返回一个由新的对象以及 `True` 组成的元组。新的对象将会大概按照以下的逻辑创建:

```
params = {k: v for k, v in kwargs.items() if '__' not in k}
params.update(defaults)
obj = self.model(**params)
obj.save()
```

它表示从非 `'defaults'` 且不包含双下划线的关键字参数开始（暗示这是一个不精确的查询）。然后将 `defaults` 的内容添加进来，覆盖必要的键，并使用结果作为关键字参数传递给模型类。这是对用到的算法的简单描述，但它包含了所有的相关的细节。内部的实现有更多的错误检查并处理一些边缘条件；如果感兴趣，请阅读代码。

如果你有一个名为 `defaults` 的字段，并且想在 `get_or_create()` 是用它作为精确查询，只需要使用 `'defaults__exact'` ，像这样：

```
Foo.objects.get_or_create(defaults__exact='bar', defaults={'defaults': 'baz'})
```

当你使用手动指定的主键时， `get_or_create()` 方法与 `create()` 方法有相似的错误行为 。如果需要创建一个对象而该对象的主键早已存在于数据库中， `IntegrityError` 异常将会被触发。

这个方法假设正确使用原子操作，正确的数据库配置和底层数据库的正确行为。然而，如果数据库级别没有对 `get_or_create` 中用到的 `kwargs` 强制要求唯一性（参见 `unique` 和 `unique_together` ），这个方法容易导致竞态条件可能会仍具有相同参数的多行同时插入。

如果你正在使用MySQL，请确保使用 `READ COMMITTED` 隔离级别而不是默认的 `REPEATABLE READ` ，否则你将会遇到 `get_or_create` 引发 `IntegrityError` 但对象在接下来的 `get()` 调用中并不存在的情况。

最后讲一句 `get_or_create()` 在Django 视图中的使用。请确保只在 `POST` 请求中使用，除非你有充分的理由。 `GET` 请求不应该对数据有任何影响。而 `POST` 则用于对数据产生影响的请求。更多信息，参见HTTP 细则中的安全的方法。

警告

你可以通过 `ManyToManyField` 属性和反向关联使用 `get_or_create()` 。在这种情况下，你应该限制查询在关联的上下文内部。如果你不一致地使用它，将可能导致完整性问题。

根据下面的模型：

```
class Chapter(models.Model):
    title = models.CharField(max_length=255, unique=True)

class Book(models.Model):
    title = models.CharField(max_length=256)
    chapters = models.ManyToManyField(Chapter)
```

你可以通过Book 的chapters 字段使用 `get_or_create()` ，但是它只会获取该Book 内部的上下文：

```
>>> book = Book.objects.create(title="Ulysses")
>>> book.chapters.get_or_create(title="Telemachus")
(<Chapter: Telemachus>, True)
>>> book.chapters.get_or_create(title="Telemachus")
(<Chapter: Telemachus>, False)
>>> Chapter.objects.create(title="Chapter 1")
<Chapter: Chapter 1>
>>> book.chapters.get_or_create(title="Chapter 1")
# Raises IntegrityError
```

发生这个错误时因为它尝试通过Book "Ulysses" 获取或者创建"Chapter 1"，但是它不能：关联关系不能获取这个chapter 因为它与这个book 不关联，但因为 `title` 字段是唯一的它仍然不能创建。

update_or_create

`update_or_create` (*defaults=None*, ***kwargs*)

New in Django 1.7.

一个通过给出的 `kwargs` 来更新对象的便捷方法， 如果需要的话创建一个新的对象。 `defaults` 是一个由 (field, value) 对组成的字典，用于更新对象。

返回一个由 `(object, created)` 组成的元组,元组中的 `object` 是一个创建的或者是被更新的对象， `created` 是一个标示是否创建了新的对象的布尔值。

`update_or_create` 方法尝试通过给出的 `kwargs` 去从数据库中获取匹配的对象。如果找到匹配的对象，它将会依据 `defaults` 字典给出的值更新字段。

这用作样板代码的一种快捷方式。例如：

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lenno
n')
    for key, value in updated_values.iteritems():
        setattr(obj, key, value)
    obj.save()
except Person.DoesNotExist:
    updated_values.update({'first_name': 'John', 'last_name': 'L
ennon'})
    obj = Person(**updated_values)
    obj.save()
```

如果模型的字段数量较大的话，这种模式就变的非常不易用。上面的示例可以用 `update_or_create()` 重写:

```
obj, created = Person.objects.update_or_create(
    first_name='John', last_name='Lennon', defaults=updated_valu
es)
```

`kwargs` 中的名称如何解析的详细描述可以参见 `get_or_create()` 。

和上文描述的 `get_or_create()` 一样，这个方式容易导致竞态条件，如果数据库层级没有前置唯一性它会让多行同时插入。

bulk_create

`bulk_create` (*objs*, *batch_size=None*)

此方法以有效的方式（通常只有1个查询，无论有多少对象）将提供的对象列表插入到数据库中：

```
>>> Entry.objects.bulk_create([
...     Entry(headline="Django 1.0 Released"),
...     Entry(headline="Django 1.1 Announced"),
...     Entry(headline="Breaking: Django is awesome")
... ])
```

这有一些注意事项：

- 将不会调用模型的 `save()` 方法，并且不会发送 `pre_save` 和 `post_save` 信号。
- 它不适用于多表继承场景中的子模型。
- 如果模型的主键是 `AutoField` ，它不会像 `save()` 那样检索和设置主键属性。
- 它不适用于多对多关系。

`batch_size` 参数控制在单个查询中创建的对象数。默认值是在一个批处理中创建所有对象，除了SQLite，其中默认值为每个查询最多使用999个变量。

count

`count` ()

返回在数据库中对应的 `QuerySet`.对象的个数。 `count()` 永远不会引发异常。

例：

```
# Returns the total number of entries in the database.
Entry.objects.count()

# Returns the number of entries whose headline contains 'Lennon'
Entry.objects.filter(headline__contains='Lennon').count()
```

`count()` 在后台执行 `SELECT COUNT（*）` `count()` ，而不是将所有的记录加载到Python对象中并在结果上调用 `len()` （除非你需要将对象加载到内存中， `len()` 会更快）。

根据您使用的数据库（例如PostgreSQL vs. MySQL）， `count()` 可能返回一个长整型而不是普通的Python整数。这是一个潜在的实现方案，不应该引起任何真实世界的问题。

请注意，如果您想要 `QuerySet` 中的项目数量，并且还要从中检索模型实例（例如，通过迭代它），使用 `len（查询集）` ，这不会导致额外的数据库查询，如 `count()` 。

in_bulk

`in_bulk` (*id_list*)

获取主键值的列表，并返回将每个主键值映射到具有给定ID的对象的实例的字典。

例：

```
>>> Blog.objects.in_bulk([1])
{1: <Blog: Beatles Blog>}
>>> Blog.objects.in_bulk([1, 2])
{1: <Blog: Beatles Blog>, 2: <Blog: Cheddar Talk>}
>>> Blog.objects.in_bulk([])
{}
```

如果你传递 `in_bulk()` 一个空列表，你会得到一个空的字典。

iterator

`iterator` ()

评估 `QuerySet` （通过执行查询），并返回一个迭代器（参见 **PEP 234**）。 `QuerySet` 通常在内部缓存其结果，以便重复计算不会导致其他查询。相反， `iterator()` 将直接读取结果，而不在 `QuerySet` 级别执行任何缓存（内部，默认迭代器调用 `iterator()` 并高速缓存返回值）。对于返回大量只需要访问一次的对象的 `QuerySet` ，这可以带来更好的性能和显着减少内存。

请注意，在已经评估的 `QuerySet` 上使用 `iterator()` 会强制它再次计算，重复查询。

此外，使用 `iterator()` 会导致先前的 `prefetch_related()` 调用被忽略，因为这两个优化一起没有意义。

警告

一些Python数据库驱动程序如 `psycopg2` 如果使用客户端游标（使用 `connection.cursor()` 实例化和Django的ORM使用）执行缓存。使用 `iterator()` 不会影响数据库驱动程序级别的缓存。要禁用此缓存，请查看服务器端游标。

latest

`latest` (*field_name=None*)

使用作为日期字段提供的 `field_name` ，按日期返回表中的最新对象。

此示例根据 `pub_date` 字段返回表中的最新 `条目` ：

```
Entry.objects.latest('pub_date')
```

如果模型的*Meta*指定 `get_latest_by` ，则可以将 `field_name` 参数留给 `earliest()` 或者 `latest()` 。默认情况下，Django将使用 `get_latest_by` 中指定的字段。

像 `get()` ， `earliest()` 和 `latest()` raise `DoesNotExist` 参数。

请注意， `earliest()` 和 `latest()` 仅仅是为了方便和可读性。

earliest

`earliest` (*field_name=None*)

除非方向更改，否则像 `latest()` 。

first

`first` ()

返回结果集的第一个对象, 当没有找到时返回 `None` .如果 `QuerySet` 没有设置排序,则将会自动按主键进行排序

例：

```
p = Article.objects.order_by('title', 'pub_date').first()
```

笔记: `first()` 是一个简便方法 下面这个例子和上面的代码效果是一样

```
try:
    p = Article.objects.order_by('title', 'pub_date')[0]
except IndexError:
    p = None
```

last

`last` ()

工作方式类似 `first()` ，只是返回的是查询集中最后一个对象。

aggregate（聚合查询）

`aggregate` (*args, **kwargs)

返回一个字典，包含根据 `QuerySet` 计算得到的聚合值（平均数、和等等）。 `aggregate()` 的每个参数指定返回的字典中将要包含的值。

Django 提供的聚合函数在下文的聚合函数中讲述。因为聚合也是查询表达式，你可以组合多个聚合以及值来创建复杂的聚合。

使用关键字参数指定的聚合将使用关键字参数的名称作为Annotation 的名称。匿名的参数的名称将基于聚合函数的名称和模型字段生成。复杂的聚合不可以使用匿名参数，它们必须指定一个关键字参数作为别名。

例如，当你使用Blog Entry 时，你可能想知道对Author 贡献的Blog Entry 的数目：

```
>>> from django.db.models import Count
>>> q = Blog.objects.aggregate(Count('entry'))
{'entry__count': 16}
```

通过使用关键字参数来指定聚合函数，你可以控制返回的聚合的值的名称：

```
>>> q = Blog.objects.aggregate(number_of_entries=Count('entry'))
{'number_of_entries': 16}
```

聚合的深入讨论，参见聚合的指南。

exists

`exists` ()

如果 `QuerySet` 包含任何结果，则返回 `True` ，否则返回 `False` 。它会试图用最简单和最快的方法完成查询，但它执行的方法与普通的*QuerySet* 查询 `确实` 几乎相同。

`exists()` 用于搜寻对象是否在 `QuerySet` 中以及 `QuerySet` 是否存在任何对象，特别是 `QuerySet` 比较大的时候。

查找具有唯一性字段（例如 `primary_key` ）的模型是否在一个 `QuerySet` 中的最高效的方法是：

```
entry = Entry.objects.get(pk=123)
if some_queryset.filter(pk=entry.pk).exists():
    print("Entry contained in queryset")
```

它将比下面的方法快很多，这个方法要求对QuerySet 求值并迭代整个QuerySet：

```
if entry in some_queryset:
   print("Entry contained in QuerySet")
```

若要查找一个QuerySet 是否包含任何元素：

```
if some_queryset.exists():
    print("There is at least one object in some_queryset")
```

将快于：

```
if some_queryset:
    print("There is at least one object in some_queryset")
```

... 但不会快很多（因为这需要很大的QuerySet 以获得效率的提升）。

另外，如果 `some_queryset` 还没有求值，但你知道它将在某个时刻求值，那么使用 `some_queryset.exists()` 将比简单地使用 `bool(some_queryset)` 完成更多的工作（一个查询用于存在性检查，另外一个是后面的求值），后者将求值并检

查是否有结果返回。

update

`update` (***kwargs*)

对指定的字段执行SQL更新查询，并返回匹配的行数（如果某些行已具有新值，则可能不等于已更新的行数）。

例如，要对2010年发布的所有博客条目启用评论，您可以执行以下操作：

```
>>> Entry.objects.filter(pub_date__year=2010).update(comments_on
=False)
```

（假设您的 `输入` 模型具有字段 `pub_date` 和 `comments_on` 。）

您可以更新多个字段 - 没有多少字段的限制。例如，在这里我们更新 `comments_on` 和 `标题` 字段：

```
>>> Entry.objects.filter(pub_date__year=2010).update(comments_on
=False, headline='This is old')
```

`update()` 方法立即应用，对更新的 `QuerySet` 的唯一限制是它只能更新模型主表中的列，而不是相关模型。你不能这样做，例如：

```
>>> Entry.objects.update(blog__name='foo') # Won't work!
```

仍然可以根据相关字段进行过滤：

```
>>> Entry.objects.filter(blog__id=1).update(comments_on=True)
```

您不能在 `QuerySet` 上调用 `update()` ，该查询已截取一个切片，或者无法再进行过滤。

`update()` 方法返回受影响的行数：

```
>>> Entry.objects.filter(id=64).update(comments_on=True)
1

>>> Entry.objects.filter(slug='nonexistent-slug').update(comment
s_on=True)
0

>>> Entry.objects.filter(pub_date__year=2010).update(comments_on
=False)
132
```

如果你只是更新一个记录，不需要对模型对象做任何事情，最有效的方法是调用 `update()` ，而不是将模型对象加载到内存中。例如，而不是这样做：

```
e = Entry.objects.get(id=10)
e.comments_on = False
e.save()
```

...做这个：

```
Entry.objects.filter(id=10).update(comments_on=False)
```

使用 `update()` 还可以防止在加载对象和调用 `save()` 之间的短时间内数据库中某些内容可能发生更改的竞争条件。

Finally, realize that `update()` does an update at the SQL level and, thus, does not call any `save()` methods on your models, nor does it emit the `pre_save` or `post_save` signals (which are a consequence of calling `Model.save()`). 如果你想更新一个具有自定义 `save()` 方法的模型的记录，请循环遍历它们并调用 `save()` ，如下所示：

```
for e in Entry.objects.filter(pub_date__year=2010):
    e.comments_on = False
    e.save()
```

delete

`delete` ()

对 `QuerySet` 中的所有行执行SQL删除查询。立即应用 `delete()` 。您不能在 `QuerySet` 上调用 `delete()` ，该查询已采取切片或以其他方式无法过滤。

例如，要删除特定博客中的所有条目：

```
>>> b = Blog.objects.get(pk=1)

# Delete all the entries belonging to this Blog.
>>> Entry.objects.filter(blog=b).delete()
```

默认情况下，Django的 `ForeignKey` 模拟SQL约束 `ON DELETE CASCADE` 字，任何具有指向要删除的对象的外键的对象将与它们一起被删除。例如：

```
blogs = Blog.objects.all()
# This will delete all Blogs and all of their Entry objects.
blogs.delete()
```

此级联行为可通过 `ForeignKey` 的 `on_delete` 参数自定义。

`delete()` 方法执行批量删除，并且不会在模型上调用任何 `delete()` 方法。但它会为所有已删除的对象（包括级联删除）发出 `pre_delete` 和 `post_delete` 信号。

Django需要获取对象到内存中以发送信号和处理级联。然而，如果没有级联和没有信号，那么Django可以采取快速路径并删除对象而不提取到内存中。对于大型删除，这可以显着减少内存使用。执行的查询量也可以减少。

设置为 `on_delete` `DO_NOTHING` 的外键不会阻止删除快速路径。

请注意，在对象删除中生成的查询是实施详细信息，可能会更改。

as_manager

classmethod `as_manager` ()

New in Django 1.7.

类方法，返回 `Manager` 的实例与 `QuerySet` 的方法的副本。有关详细信息，请参见*Creating Manager with QuerySet methods*。

字段查找

字段查询是指如何指定SQL `WHERE` 子句的内容. 它们通过 `查询集` 的 `filter()`, `exclude()` and `get()` 的关键字参数指定.

查阅简介, 请参考 模型与数据库查询.

Django的内置查找列在下面。也可以为模型字段写入*custom lookups*。

为了方便当没有提供查找类型时（例如 `Entry.objects.get(id=14)` ），假设查找类型为 `exact` 。

exact

精确匹配。如果为比较提供的值为 `None` ，它将被解释为SQL `NULL` （有关详细信息，请参阅 `isnull` ）。

例子：

```
Entry.objects.get(id__exact=14)
Entry.objects.get(id__exact=None)
```

SQL等价物：

```
SELECT ... WHERE id = 14;
SELECT ... WHERE id IS NULL;
```

MySQL比较

在MySQL中，数据库表的“排序规则”设置确定 `exact` 比较是否区分大小写。这是一个数据库设置，而不是一个Django设置。可以配置MySQL表以使用区分大小写的比较，但涉及一些折衷。有关详细信息，请参阅*databases*文档中的*collation section*。

iexact

不区分大小写的精确匹配

Changed in Django 1.7:

如果为比较提供的值为 `None` ，它将被解释为SQL `NULL` （有关详细信息，请参阅 `isnull` ）。

例：

```
Blog.objects.get(name__iexact='beatles blog')
Blog.objects.get(name__iexact=None)
```

SQL等价物：

```
SELECT ... WHERE name ILIKE 'beatles blog';
SELECT ... WHERE name IS NULL;
```

请注意，第一个查询将匹配 `'Beatles Blog'` , `'beatles blog'` , `'BeAtLes BLoG'` , etc.

SQLite用户

当使用SQLite后端和Unicode（非ASCII）字符串时，请记住关于字符串比较的数据库注释。SQLite不对Unicode字符串进行不区分大小写的匹配。

contains

区分敏感遏制试验。

例：

```
Entry.objects.get(headline__contains='Lennon')
```

SQL等效：

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

请注意，这将匹配标题 `'Lennon honored today'` ，但不符合 `'lennon honored today'` .

SQLite用户

SQLite不支持区分大小写的 `LIKE` 语句； `contains` 用于SQLite的 `icontains` 。有关详细信息，请参阅数据库注释。

icontains

不区分大小写的遏制试验。

例：

```
Entry.objects.get(headline__icontains='Lennon')
```

SQL等效：

```
SELECT ... WHERE headline ILIKE '%Lennon%';
```

SQLite用户

当使用SQLite后端和Unicode（非ASCII）字符串时，请记住关于字符串比较的数据库注释。

in

在给定的列表。

例：

```
Entry.objects.filter(id__in=[1, 3, 4])
```

SQL等效：

```
SELECT ... WHERE id IN (1, 3, 4);
```

您还可以使用查询集动态评估值列表，而不是提供文字值列表：

```
inner_qs = Blog.objects.filter(name__contains='Cheddar')
entries = Entry.objects.filter(blog__in=inner_qs)
```

此查询集将作为subselect语句求值：

```
SELECT ... WHERE blog.id IN (SELECT id FROM ... WHERE NAME LIKE
'%Cheddar%')
```

如果您传入 `ValuesQuerySet` 或 `ValuesListQuerySet` （调用 `values()` 或 `values_list()` 查询集）作为 `__在` 查找的值，您需要确保您只提取结果中的一个字段。例如，这将工作（过滤博客名称）：

```
inner_qs = Blog.objects.filter(name__contains='Ch').values('name
')
entries = Entry.objects.filter(blog__name__in=inner_qs)
```

这个例子将产生一个异常，由于内查询试图提取两个字段的值，但是查询语句只期望提取一个字段的值：

```
# Bad code! Will raise a TypeError.
inner_qs = Blog.objects.filter(name__contains='Ch').values('name
', 'id')
entries = Entry.objects.filter(blog__name__in=inner_qs)
```

性能注意事项

对于使用嵌套查询和了解数据库服务器的性能特征（如果有疑问，去做基准测试）要谨慎。一些数据库后端，最着名的是MySQL，不能很好地优化嵌套查询。在这些情况下，提取值列表然后将其传递到第二个查询中更有效。也就是说，执行两个查询，而不是一个：

```
values = Blog.objects.filter(
        name__contains='Cheddar').values_list('pk', flat=True)
entries = Entry.objects.filter(blog__in=list(values))
```

请注意`list()`调用Blog `QuerySet`以强制执行第一个查询。没有它，将执行嵌套查询，因为QuerySet是惰性的。

）

大于

例子:

```
Entry.objects.filter(id__gt=4)
```

SQL语句相当于：

```
SELECT ... WHERE id > 4;
```

gte

大于或等于

1

小于

lte

小于或等于

startswith

区分大小写，开始位置匹配

例：

```
Entry.objects.filter(headline__startswith='Will')
```

SQL等效：

```
SELECT ... WHERE headline LIKE 'Will%';
```

SQLite 不支持区分大小写 `LIKE` 语句; Sqlite 下 `startswith` 等于 `istartswith` .

istartswith

不区分大小写，开始位置匹配

例：

```
Entry.objects.filter(headline__istartswith='will')
```

SQL等效：

```
SELECT ... WHERE headline ILIKE 'Will%';
```

SQLite用户

当使用SQLite后端和Unicode（非ASCII）字符串时，请记住关于字符串比较的数据库注释。

endswith

区分大小写。

例：

```
Entry.objects.filter(headline__endswith='cats')
```

SQL等效：

```
SELECT ... WHERE headline LIKE '%cats';
```

SQLite用户

SQLite不支持区分大小写的 `LIKE` 语句； `endswith` 用作SQLite的 `iendswith` 。有关更多信息，请参阅数据库注释文档。

iendswith

不区分大小写。

例：

```
Entry.objects.filter(headline__iendswith='will')
```

SQL等效：

```
SELECT ... WHERE headline ILIKE '%will'
```

SQLite用户

当使用SQLite后端和Unicode（非ASCII）字符串时，请记住关于字符串比较的数据库注释。

range

范围测试（包含于之中）。

例：

```
import datetime
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
Entry.objects.filter(pub_date__range=(start_date, end_date))
```

SQL等效：

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' and '2005-03-31';
```

您可以在任何可以使用 `BETWEEN` 的SQL中使用 `范围` （对于日期，数字和偶数字符）。

警告

过滤具有日期的 `DateTimeField` 不会包含最后一天的项目，因为边界被解释为“给定日期的0am”。如果 `pub_date` 是 `DateTimeField` ，上面的表达式将变成这个SQL：

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01 00:00:00' and '200
5-03-31 00:00:00';
```

一般来说，不能混合使用日期和数据时间。

year

对于日期和日期时间字段，确切的年匹配。整数年。

例：

```
Entry.objects.filter(pub_date__year=2005)
```

SQL等效：

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' AND '2005-12-31';
```

（确切的SQL语法因每个数据库引擎而异）。

当 `USE_TZ` 为 `True` 时，在过滤之前，datetime字段将转换为当前时区。

month

对于日期和日期时间字段，确切的月份匹配。取整数1（1月）至12（12月）。

例：

```
Entry.objects.filter(pub_date__month=12)
```

SQL等效：

```
SELECT ... WHERE EXTRACT('month' FROM pub_date) = '12';
```

（确切的SQL语法因每个数据库引擎而异）。

当 `USE_TZ` 为 `True` 时，在过滤之前，datetime字段将转换为当前时区。这需要数据库中的时区定义。

day(要写到程序里的字段麻烦不要自作聪明翻译谢谢)

对于日期和日期时间字段，具体到某一天的匹配。取一个整数的天数。

例：

```
Entry.objects.filter(pub_date__day=3)
```

SQL等效：

```
SELECT ... WHERE EXTRACT('day' FROM pub_date) = '3';
```

（确切的SQL语法因每个数据库引擎而异）。

请注意，这将匹配每月第三天（例如1月3日，7月3日等）的任何包含pub_date的记录。

当 `USE_TZ` 为 `True` 时，在过滤之前，datetime字段将转换为当前时区。这需要数据库中的*time zone definitions in the database*。

week_day

对于日期和日期时间字段，“星期几”匹配。

取整数值，表示星期几从1（星期日）到7（星期六）。

例：

```
Entry.objects.filter(pub_date__week_day=2)
```

（此查找不包括等效的SQL代码片段，因为相关查询的实现因不同数据库引擎而异）。

请注意，这将匹配落在星期一（星期二）的任何记录（ `pub_date` ），而不管其出现的月份或年份。周日被索引，第1天为星期天，第7天为星期六。

当 `USE_TZ` 为 `True` 时，在过滤之前，datetime字段将转换为当前时区。这需要数据库中的时区定义。

hour

对于日期时间字段，精确的小时匹配。取0和23之间的整数。

例：

```
Event.objects.filter(timestamp__hour=23)
```

SQL等效：

```
SELECT ... WHERE EXTRACT('hour' FROM timestamp) = '23';
```

（确切的SQL语法因每个数据库引擎而异）。

当 `USE_TZ` 为 `True` 时，在过滤之前将值转换为当前时区。

minute

对于日期时间字段，精确的分钟匹配。取0和59之间的整数。

例：

```
Event.objects.filter(timestamp__minute=29)
```

SQL等效：

```
SELECT ... WHERE EXTRACT('minute' FROM timestamp) = '29';
```

（确切的SQL语法因每个数据库引擎而异）。

当 `USE_TZ` 为 `True` 时，在过滤之前将值转换为当前时区。

second

对于datetime字段，精确的第二个匹配。取0和59之间的整数。

例：

```
Event.objects.filter(timestamp__second=31)
```

等同于SQL语句:

```
SELECT ... WHERE EXTRACT('second' FROM timestamp) = '31';
```

（确切的SQL语法因每个数据库引擎而异）。

当 `USE_TZ` 为 `True` 时，在过滤之前将值转换为当前时区。

isnull

值为 `True` 或 `False` , 相当于 SQL语句 `IS NULL` 和 `IS NOT NULL` .

例：

```
Entry.objects.filter(pub_date__isnull=True)
```

SQL等效：

```
SELECT ... WHERE pub_date IS NULL;
```

search

一个Boolean类型的全文搜索，以全文搜索的优势。这个很像 `contains` ，但是由于全文索引的优势，以使它更显著的快。

例：

```
Entry.objects.filter(headline__search="+Django -jazz Python")
```

SQL等效：

```
SELECT ... WHERE MATCH(tablename, headline) AGAINST (+Django -jazz Python IN BOOLEAN MODE);
```

注意，这仅在MySQL中可用，并且需要直接操作数据库以添加全文索引。默认情况下，Django使用BOOLEAN MODE进行全文搜索。有关其他详细信息，请参阅MySQL文档。

正则表达式

区分大小写的正则表达式匹配。

正则表达式语法是正在使用的数据库后端的语法。在SQLite没有内置正则表达式支持的情况下，此功能由（Python）用户定义的REGEXP函数提供，因此正则表达式语法是Python的 `re` 模块。

例：

```
Entry.objects.get(title__regex=r'^(An?|The) +')
```

SQL等价物：

```
SELECT ... WHERE title REGEXP BINARY '^(An?|The) +'; -- MySQL

SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'c'); -- Oracle

SELECT ... WHERE title ~ '^(An?|The) +'; -- PostgreSQL

SELECT ... WHERE title REGEXP '^(An?|The) +'; -- SQLite
```

建议使用原始字符串（例如， `r'foo'` 而不是 `'foo'` ）来传递正则表达式语法。

iregex

不区分大小写的正则表达式匹配。

例：

```
Entry.objects.get(title__iregex=r'^(an?|the) +')
```

SQL等价物：

```
SELECT ... WHERE title REGEXP '^(an?|the) +'; -- MySQL

SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'i'); -- Ora
cle

SELECT ... WHERE title ~* '^(an?|the) +'; -- PostgreSQL

SELECT ... WHERE title REGEXP '(?i)^(an?|the) +'; -- SQLite
```

聚合函数

Django 的 `django.db.models` 模块提供以下聚合函数。关于如何使用这些聚合函数的细节，参见聚合函数的指南。关于如何创建聚合函数，参数 `聚合函数` 的文档。

警告

SQLite 不能直接处理日期/时间字段的聚合。这是因为SQLite 中没有原生的日期/时间字段，Django 目前使用文本字段模拟它的功能。在SQLite 中对日期/时间字段使用聚合将引发 `NotImplementedError` 。

注

在 `QuerySet` 为空时，聚合函数函数将返回 `None` 。例如，如果 `QuerySet` 中没有记录， `Sum` 聚合函数将返回 `None` 而不是 `0` 。 `Count` 是一个例外，如果 `QuerySet` 为空，它将返回 `0` 。

所有聚合函数具有以下共同的参数：

表达

引用模型字段的一个字符串，或者一个查询表达式。

New in Django 1.8:

现在在复杂的计算中，聚合函数可以引用多个字段。

output_field

用来表示返回值的模型字段，它是一个可选的参数。

New in Django 1.8:

添加 `output_field` 参数。

注

在组合多个类型的字段时，只有在所有的字段都是相同类型的情况下，Django 才能确定 `output_field` 。否则，你必须自己提供 `output_field` 参数。

**额外

这些关键字参数可以给聚合函数生成的SQL 提供额外的信息。

avg

class `Avg` (*expression*, *output_field=None*, ***extra*)

返回给定expression 的平均值，其中expression 必须为数值。

- 默认的别名： <field>__avg
- 返回类型： `float`

Count

class `Count` (*expression*, *distinct=False*, ***extra*)

返回与expression 相关的对象的个数。

- 默认的别名： <field>__count
- 返回类型： `int`

有一个可选的参数：

`distinct`

如果 `distinct=True` ，Count 将只计算唯一的实例。它等同于 COUNT(DISTINCT <field>) SQL 语句。默认值为 `False` 。

Max

class `Max` (*expression*, *output_field=None*, ***extra*)

返回expression 的最大值。

- 默认的别名： <field>__max
- 返回类型：与输入字段的类型相同，如果提供则为 `output_field` 类型

Min

class `Min` (*expression*, *output_field=None*, ***extra*)

返回expression 的最小值。

- 默认的别名：`<field>__min`
- 返回的类型：与输入字段的类型相同，如果提供则为 `output_field` 类型

StdDev

class `StdDev` (*expression*, *sample=False*, ***extra*)

返回expression 的标准差。

- 默认的别名：`<field>__stddev`
- 返回类型：`float`

有一个可选的参数：

`sample`

默认情况下，`StdDev` 返回群体的标准差。但是，如果 `sample=True`，返回的值将是样本的标准差。

SQLite

SQLite 没有直接提供 `StdDev` 。有一个可用的实现是SQLite 的一个扩展模块。参见SQlite 的文档 中获取并安装这个扩展的指南。

Sum

class `Sum` (*expression*, *output_field=None*, ***extra*)

计算expression 的所有值的和。

- 默认的别名：`<field>__sum`
- 返回类型：与输入的字段相同，如果提供则为 `output_field` 的类型

Variance

class `Variance` (*expression*, *sample=False*, ***extra*)

返回expression 的方差。

- 默认的别名：`<field>__variance`
- 返回的类型：`float`

有一个可选的参数：

`sample`

默认情况下，`Variance` 返回群体的方差。但是，如果`sample=True` ，返回的值将是样本的方差。

SQLite

SQLite 没有直接提供`Variance` 。有一个可用的实现是SQLite 的一个扩展模块。参见SQlite 的文档 中获取并安装这个扩展的指南。

查询相关的类

本节提供查询相关的工具的参考资料，它们其它地方没有文档。

`Q()` 对象

class `Q`

`Q()` 对象和`F` 对象类似，把一个SQL 表达式封装在Python 对象中，这个对象可以用于数据库相关的操作。

通常，`Q() 对象` 使得定义查询条件然后重用成为可能。这允许*construction of complex database queries*使用 `|` (`OR`) 和 `&` (`AND`) 操作符; 否则 `QuerySets` 中使用不了 `OR` 。

`Prefetch()` 对象

New in Django 1.7.

class `Prefetch` (*lookup*, *queryset=None*, *to_attr=None*)

`Prefetch()` 对象可用于控制 `prefetch_related()` 的操作。

`lookup` 参数描述了跟随的关系，并且工作方式与传递给 `prefetch_related()` 的基于字符串的查找相同。例如：

```
>>> Question.objects.prefetch_related(Prefetch('choice_set')).ge
t().choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking ag
ain>]
# This will only execute two queries regardless of the number of
 Question
# and Choice objects.
>>> Question.objects.prefetch_related(Prefetch('choice_set')).al
l()
[<Question: Question object>]
```

`查询集` 参数为给定的查找提供基本 `QuerySet` 。这对于进一步过滤预取操作或从预取关系调用 `select_related()` 很有用，因此进一步减少查询数量：

```
>>> voted_choices = Choice.objects.filter(votes__gt=0)
>>> voted_choices
[<Choice: The sky>]
>>> prefetch = Prefetch('choice_set', queryset=voted_choices)
>>> Question.objects.prefetch_related(prefetch).get().choice_set
.all()
[<Choice: The sky>]
```

`to_attr` 参数将预取操作的结果设置为自定义属性：

```
>>> prefetch = Prefetch('choice_set', queryset=voted_choices, to
_attr='voted_choices')
>>> Question.objects.prefetch_related(prefetch).get().voted_choi
ces
[<Choice: The sky>]
>>> Question.objects.prefetch_related(prefetch).get().choice_set
.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking ag
ain>]
```

注意

当使用 `to_attr` 时，预取的结果存储在列表中。这可以提供比存储在 `QuerySet` 实例内的缓存结果的传统 `prefetch_related` 调用显着的速度改进。

查找 API 参考

```
New in Django 1.7.
```

这篇文档是查找 API 的参考，Django 用这些API 构建数据库查询的 `WHERE` 子句。若要学习如何使用 查找，参见执行查询；若要了解如何创建 新的查找，参见自定义查找。

查找 API 由两个部分组成：`RegisterLookupMixin` 类，它用于注册查找；查询表达式API，它是一个方法集，类必须实现它们才可以注册成一个查找。

Django 有两个类遵循查询表达式API，且Django 所有内建的查找都继承自它们：

- `Lookup`：用于查找一个字段（例如 `field_name__exact` 中的 `exact`）
- `Transform`：用于转换一个字段

查找表达式由三部分组成：

- 字段部分（例如，`Book.objects.filter(author__best_friends__first_name...)`；
- 转换部分（可以省略）（例如，`__lower__first3chars__reversed`）；
- 查找部分（例如，`__icontains`），如果省略则默认为 `__exact`。

注册 API

Django 使用 `RegisterLookupMixin` 来为类提供接口，注册它自己的查找。两个最突出的例子是 `Field`（所有模型字段的基类）和 `Aggregate`（Django 所有聚合函数的基类）。

`class lookups.RegisterLookupMixin`

一个mixin，实现一个类上的查找API。

`classmethod register_lookup(lookup)`

在类中注册一个新的查找。例如，`DateField.register_lookup(YearExact)` 将在 `DateField` 上注册一个 `YearExact` 查找。它会覆盖已存在的同名查找。

`get_lookup(lookup_name)`

返回类中注册的名为 `lookup_name` 的 `Lookup`。默认的实现会递归查询所有的父类，并检查它们中的任何一个是否具有名称为 `lookup_name` 的查找，并返回第一个匹配。

`get_transform(transform_name)`

返回一个名为 `transform_name` 的 `Transform` 。默认的实现会递归查找所有的父类，并检查它们中的任何一个是否具有名称为 `transform_name` 的查找，并返回第一个匹配。

一个类如果想要成为查找，它必须实现查询表达式API。 `Lookup` 和 `Transform` 一开始就遵循这个API。

查询表达式API

查询表达式API是一个通用的方法集，在查询表达式中可以使用定义了这些方法的类，来将它们自身转换为SQL表达式。直接的字段引用，聚合，以及 `Transform` 类都是遵循这个API的示例。当一个对象实现以下方法时，就被称为遵循查询表达式API：

`as_sql(self, compiler, connection)`

负责从表达式中产生查询字符串和参数。 `compiler` 是一个 `SQLCompiler` 对象，它拥有可以编译其它表达式的 `compile()` 方法。 `connection` 是用于执行查询的连接。

调用 `expression.as_sql()` 一般是不对的 -- 而是应该调用 `compiler.compile(expression)` 。 `compiler.compile()` 方法应该在调用表达式的供应商特定方法时格外小心。

`as_vendorname(self, compiler, connection)`

和 `as_sql()` 的工作方式类似。当一个表达式经过 `compiler.compile()` 编译之后， Django会首先尝试调用 `as_vendorname()` ，其中 `vendorname` 是用于执行查询的后端供应商。对于Django内建的后端， `vendorname` 是 `postgresql` ， `oracle` ， `sqlite` ，或者 `mysql` 之一。

`get_lookup(lookup_name)`

必须返回名称为 `lookup_name` 的查找。例如，通过返回 `self.output_field.get_lookup(lookup_name)` 来实现。

`get_transform(transform_name)`

必须返回名称为 `transform_name的` 查找。例如，通过返回 `self.output_field.get_transform(transform_name)` 来实现。

`output_field`

定义 `get_lookup()` 方法所返回的类的类型。必须为 `Field` 的实例。

Transform 类参考

`class Transform`

Transform 是用于实现字段转换的通用类。一个显然的例子是 __year 会把 DateField 转换为 IntegerField 。

在表达式中执行查找的标记是 Transform<expression>__<transformation> (例如 date__year)。

这个类遵循查询表达式API，也就是说你可以使用 <expression>__<transform1>__<transform2> 。

bilateral

```
New in Django 1.8.
```

一个布尔值，表明是否对 lhs 和 rhs 都应用这个转换。如果对两侧都应用转换，应用在 rhs 的顺序和在查找表达式中的出现顺序相同。默认这个属性为 False 。使用方法的实例请见自定义查找。

lhs

在左边，也就是被转换的东西。必须遵循查询表达式API。

lookup_name

查找的名称，用于在解析查询表达式的时候识别它。

output_field

为这个类定义转换后的输出。必须为 Field 的实例。默认情况下和 lhs.output_field 相同。

as_sql()

需要被覆写；否则抛出 NotImplementedError 异常。

get_lookup(lookup_name)

和 get_lookup() 相同。

get_transform(transform_name)

和 get_transform() 相同。

Lookup 类参考

class Lookup

Lookup 是实现查找的通用的类。查找是一个查询表达式，它的左边是 lhs ，右边是 rhs ； lookup_name 用于构造 lhs 和 rhs 之间的比较，来产生布尔值，例如 lhs in rhs 或者 lhs > rhs 。

在表达式中执行查找的标记是 <lhs>__<lookup_name>=<rhs> 。

这个类并不遵循查询表达式API，因为在它构造的时候出现了 `=<rhs>` ：查找总是在查找表达式的最后。

`lhs`

在左边，也就是被查找的东西。这个对象必须遵循查询表达式API。

`rhs`

在右边，也就是用来和 `lhs` 比较的东西。它可以是个简单的值，也可以是在SQL中编译的一些东西，比如 `F()` 对象或者 `QuerySet` 。

`lookup_name`

查找的名称，用于在解析查询表达式的时候识别它。

`process_lhs(compiler, connection[, lhs=None])`

返回元组 `(lhs_string, lhs_params)` ，和 `compiler.compile(lhs)` 所返回的一样。这个方法可以被覆写，来调整 `lhs` 的处理方式。

`compiler` 是一个 `SQLCompiler` 对象，可以像 `compiler.compile(lhs)` 这样使用来编译 `lhs` 。 `connection` 可以用于编译供应商特定的SQL语句。 `lhs` 如果不为 `None` , 会代替 `self.lhs` 作为处理后的 `lhs` 使用。

`process_rhs(compiler, connection)`

对于右边的东西，和 `process_lhs()` 的行为相同。

> 译者：Django 文档协作翻译小组，原文：Lookup expressions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

模型的实例

模型实例参考

该文档详细描述模型 的API。它建立在模型 和执行查询 的资料之上， 所以在阅读这篇文档之前，你可能会想要先阅读并理解那两篇文档。

我们将用执行查询中所展现的 博客应用模型 来贯穿这篇参考文献。

创建对象

要创建模型的一个新实例，只需要像其它Python 类一样实例化它：

`class Model(**kwargs)`

关键字参数就是在你的模型中定义的字段的名字。注意，实例化一个模型不会访问数据库；若要保存，你需要save() 一下。

> 注
>
> 也许你会想通过重写 `__init__` 方法来自定义模型。无论如何，如果你这么做了，小心不要改变了调用签名——任何改变都可能阻碍模型实例被保存。尝试使用下面这些方法之一，而不是重写**init**：
>
> 1. 在模型类中增加一个类方法：

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)

    @classmethod
    def create(cls, title):
        book = cls(title=title)
        # do something with the book
        return book

book = Book.create("Pride and Prejudice")
```

> 2. 在自定义管理器中添加一个方法（推荐）：

```
class BookManager(models.Manager):
    def create_book(self, title):
        book = self.create(title=title)
        # do something with the book
        return book

class Book(models.Model):
    title = models.CharField(max_length=100)

    objects = BookManager()

book = Book.objects.create_book("Pride and Prejudice")
```

自定义模型加载

`classmethod Model.from_db(db, field_names, values)`

```
New in Django 1.8.
```

`from_db()` 方法用于自定义从数据库加载时模型实例的创建。

`db` 参数包含数据库的别名，`field_names` 包含所有加载的字段的名称，`values` 包含 `field_names` 中每个字段加载的值。`field_names` 与 `values` 的顺序相同，所以可以使用 `cls(**(zip(field_names, values)))` 来实例化对象。如果模型的所有字段都提供，会保证 `values` 的顺序与 `__init__()` 所期望的一致。这表示此时实例可以通过 `cls(*values)` 创建。可以通过 `cls._deferred` 来检查是否提供所有的字段 —— 如果为 `False`，那么所有的字段都已经从数据库中加载。

除了创建新模型之前，`from_db()` 必须设置新实例 `_state` 属性中的 `adding` 和 `db` 标志位。

下面的示例演示如何保存从数据库中加载进来的字段原始值：

```
@classmethod
def from_db(cls, db, field_names, values):
    # default implementation of from_db() (could be replaced
    # with super())
    if cls._deferred:
        instance = cls(**zip(field_names, values))
    else:
        instance = cls(*values)
    instance._state.adding = False
    instance._state.db = db
    # customization to store the original field values on the in
stance
    instance._loaded_values = zip(field_names, values)
    return instance

def save(self, *args, **kwargs):
    # Check how the current values differ from ._loaded_values.
For example,
    # prevent changing the creator_id of the model. (This exampl
e doesn't
    # support cases where 'creator_id' is deferred).
    if not self._state.adding and (
            self.creator_id != self._loaded_values['creator_id']
):
        raise ValueError("Updating the value of creator isn't al
lowed")
    super(...).save(*args, **kwargs)
```

上面的示例演示 `from_db()` 的完整实现。当然在这里的 `from_db()` 中完全可以只用 `super()` 调用。

从数据库更新对象

`Model.refresh_from_db(using=None, fields=None, **kwargs)`

```
New in Django 1.8.
```

如果你需要从数据库重新加载模型的一个值，你可以使用 `refresh_from_db()` 方法。当不带参数调用这个方法时，将完成以下的动作：

模型的所有非延迟字段都更新成数据库中的当前值。 之前加载的关联实例，如果关联的值不再合法，将从重新加载的实例中删除。例如，如果重新加载的实例有一个外键到另外一个模型 `Author` ，那么如果 `obj.author_id != obj.author.id` ， `obj.author` 将被扔掉并在下次访问它时根据 `obj.author_id` 的值重新加载。 注意，只有本模型的字段会从数据库重新加载。其它依赖数据库的值不会重新加载，例如聚合的结果。

重新加载使用的数据库与实例加载时使用的数据库相同，如果实例不是从数据库加载的则使用默认的数据库。可以使用 `using` 参数来强制指定重新加载的数据库。

可以回使用 `fields` 参数强制设置加载的字段。

例如，要测试 `update()` 调用是否得到预期的更新，可以编写类似下面的测试：

```
def test_update_result(self):
    obj = MyModel.objects.create(val=1)
    MyModel.objects.filter(pk=obj.pk).update(val=F('val') + 1)
    # At this point obj.val is still 1, but the value in the dat
abase
    # was updated to 2. The object's updated value needs to be r
eloaded
    # from the database.
    obj.refresh_from_db()
    self.assertEqual(obj.val, 2)
```

注意，当访问延迟的字段时，延迟字段的加载会通过这个方法加载。所以可以自定义延迟加载的行为。下面的实例演示如何在重新加载一个延迟字段时重新加载所有的实例字段：

```
class ExampleModel(models.Model):
    def refresh_from_db(self, using=None, fields=None, **kwargs)
:
        # fields contains the name of the deferred field to be
        # loaded.
        if fields is not None:
            fields = set(fields)
            deferred_fields = self.get_deferred_fields()
            # If any deferred field is going to be loaded
            if fields.intersection(deferred_fields):
                # then load all of them
                fields = fields.union(deferred_fields)
        super(ExampleModel, self).refresh_from_db(using, fields,
 **kwargs)
```

`Model.get_deferred_fields()`

```
New in Django 1.8.
```

一个辅助方法，它返回一个集合，包含模型当前所有延迟字段的属性名称。

验证对象

验证一个模型涉及三个步骤：

1. 验证模型的字段 —— `Model.clean_fields()`
2. 验证模型的完整性 —— `Model.clean()`
3. 验证模型的唯一性 —— `Model.validate_unique()`

当你调用模型的 `full_clean()` 方法时，这三个方法都将执行。

当你使用 `ModelForm` 时， `is_valid()` 将为表单中的所有字段执行这些验证。更多信息参见 `ModelForm` 文档。 如果你计划自己处理验证出现的错误，或者你已经将需要验证的字段从 `ModelForm` 中去除掉，你只需调用模型的 `full_clean()` 方法。

`Model.full_clean(exclude=None, validate_unique=True)`

该方法按顺序调用 `Model.clean_fields()` 、 `Model.clean()` 和 `Model.validate_unique()` （如果 `validate_unique` 为 `True` ），并引发一个 `ValidationError` ，该异常的 `message_dict` 属性包含三个步骤的所有错误。

可选的 `exclude` 参数用来提供一个可以从验证和清除中排除的字段名称的列表。 `ModelForm` 使用这个参数来排除表单中没有出现的字段，使它们不需要验证，因为用户无法修正这些字段的错误。

注意，当你调用模型的 `save()` 方法时， `full_clean()` 不会 自动调用。如果你想一步就可以为你手工创建的模型运行验证，你需要手工调用它。例如：

```
from django.core.exceptions import ValidationError
try:
    article.full_clean()
except ValidationError as e:
    # Do something based on the errors contained in e.message_di
ct.
    # Display them to a user, or handle them programmatically.
    pass
```

`full_clean()` 第一步执行的是验证每个字段。

`Model.clean_fields(exclude=None)`

这个方法将验证模型的所有字段。可选的 `exclude` 参数让你提供一个字段名称列表来从验证中排除。如果有字段验证失败，它将引发一个 `ValidationError` 。

`full_clean()` 第二步执行的是调用 `Model.clean()` 。如要实现模型自定义的验证，应该覆盖这个方法。

`Model.clean()`

应该用这个方法来提供自定义的模型验证，以及修改模型的属性。例如，你可以使用它来给一个字段自动提供值，或者用于多个字段需要一起验证的情形：

```
import datetime
from django.core.exceptions import ValidationError
from django.db import models

class Article(models.Model):
    ...
    def clean(self):
        # Don't allow draft entries to have a pub_date.
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError('Draft entries may not have a
publication date.')
        # Set the pub_date for published items if it hasn't been
 set already.
        if self.status == 'published' and self.pub_date is None:
            self.pub_date = datetime.date.today()
```

然而请注意，和 `Model.full_clean()` 类似，调用模型的 `save()` 方法时不会引起 `clean()` 方法的调用。

在上面的示例中， `Model.clean()` 引发的 `ValidationError` 异常通过一个字符串实例化，所以它将被保存在一个特殊的错误字典键 `NON_FIELD_ERRORS` 中。这个键用于整个模型出现的错误而不是一个特定字段出现的错误：

```
from django.core.exceptions import ValidationError, NON_FIELD_ER
RORS
try:
    article.full_clean()
except ValidationError as e:
    non_field_errors = e.message_dict[NON_FIELD_ERRORS]
```

若要引发一个特定字段的异常，可以使用一个字典实例化 `ValidationError` ，其中字典的键为字段的名称。我们可以更新前面的例子，只引发 `pub_date` 字段上的异常：

```
class Article(models.Model):
    ...
    def clean(self):
        # Don't allow draft entries to have a pub_date.
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError({'pub_date': 'Draft entries ma
y not have a publication date.'})
        ...
```

最后， `full_clean()` 将检查模型的唯一性约束。

`Model.validate_unique(exclude=None)`

该方法与 `clean_fields()` 类似，只是验证的是模型的所有唯一性约束而不是单个字段的值。可选的 `exclude` 参数允许你提供一个字段名称的列表来从验证中排除。如果有字段验证失败，将引发一个 `ValidationError` 。

注意，如果你提供一个 `exclude` 参数给 `validate_unique()` ，任何涉及到其中一个字段的 `unique_together` 约束将不检查。

对象保存

将一个对象保存到数据库，需要调用 `save()` 方法：

```
Model.save([force_insert=False, force_update=False, using=DEFAULT_DI
```

如果你想要自定义保存的动作，你可以重写 `save()` 方法。请看 重写预定义的模型方法 了解更多细节。

模型保存过程还有一些细节的地方要注意；请看下面的章节。

自增的主键

如果模型具有一个 `AutoField` —— 一个自增的主键 —— 那么该自增的值将在第一次调用对象的 `save()` 时计算并保存：

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.'
)
>>> b2.id     # Returns None, because b doesn't have an ID yet.
>>> b2.save()
>>> b2.id     # Returns the ID of your new object.
```

在调用 `save()` 之前无法知道ID 的值，因为这个值是通过数据库而不是Django 计算。

为了方便，默认情况下每个模型都有一个 `AutoField` 叫做 `id` ，除非你显式指定模型某个字段的 `primary_key=True` 。更多细节参见 `AutoField` 的文档。

pk 属性

`Model.pk`

无论你是自己定义还是让Django 为你提供一个主键字段， 每个模型都将具有一个属性叫做 `pk` 。它的行为类似模型的一个普通属性，但实际上是模型主键字段属性的别名。你可以读取并设置它的值，就和其它属性一样，它会更新模型中正确的值。

显式指定自增主键的值

如果模型具有一个 `AutoField` ，但是你想在保存时显式定义一个新的对象 `ID` ，你只需要在保存之前显式指定它而不用依赖 `ID` 自动分配的值：

```
>>> b3 = Blog(id=3, name='Cheddar Talk', tagline='Thoughts on ch
eese.')
>>> b3.id     # Returns 3.
>>> b3.save()
>>> b3.id     # Returns 3.
```

如果你手工赋值一个自增主键的值，请确保不要使用一个已经存在的主键值！如果你使用数据库中已经存在的主键值创建一个新的对象，Django 将假设你正在修改这个已存在的记录而不是创建一个新的记录。

接着上面的' `Cheddar Talk` '博客示例，下面这个例子将覆盖数据库中之前的记录：

```
b4 = Blog(id=3, name='Not Cheddar', tagline='Anything but cheese
.')
b4.save()  # Overrides the previous blog with ID=3!
```

出现这种情况的原因，请参见下面的Django 如何知道是UPDATE 还是INSERT。

显式指定自增主键的值对于批量保存对象最有用，但你必须有信心不会有主键冲突。

当你保存时，发生了什么？

当你保存一个对象时，Django 执行以下步骤：

1. 发出一个 `pre-save` 信号。 发送一个 `django.db.models.signals.pre_save` 信号，以允许监听该信号的函数完成一些自定义的动作。

2. 预处理数据。 如果需要，对对象的每个字段进行自动转换。

大部分字段不需要预处理 —— 字段的数据将保持原样。预处理只用于具有特殊行为的字段。例如，如果你的模型具有一个 `auto_now=True` 的 `DateField` ，那么预处理阶段将修改对象中的数据以确保该日期字段包含当前的时间戳。（我们的文档还没有所有具有这种“特殊行为”字段的一个列表。）

3. 准备数据库数据。 要求每个字段提供的当前值是能够写入到数据库中的类型。

大部分字段不需要数据准备。简单的数据类型，例如整数和字符串，是可以直接写入的Python 对象。但是，复杂的数据类型通常需要一些改动。

例如， `DateField` 字段使用Python 的 `datetime` 对象来保存数据。数据库保存的不是 `datetime` 对象，所以该字段的值必须转换成ISO兼容的日期字符串才能插入到数据库中。

4. 插入数据到数据库中。 将预处理过、准备好的数据组织成一个SQL 语句用于插入数据库。

5. 发出一个post-save 信号。 发送一个`django.db.models.signals.post_save`信号，以允许监听听信号的函数完成一些自定义的动作。

Django 如何知道是UPDATE 还是INSERT

你可能已经注意到Django 数据库对象使用同一个`save()`方法来创建和改变对象。Django 对`INSERT`和`UPDATE` SQL 语句的使用进行抽象。当你调用`save()`时，Django 使用下面的算法：

- 如果对象的主键属性为一个求值为`True`的值（例如，非`None`值或非空字符串），Django 将执行`UPDATE`。
- 如果对象的主键属性没有设置或者`UPDATE`没有更新任何记录，Django 将执行`INSERT`。

现在应该明白了，当保存一个新的对象时，如果不能保证主键的值没有使用，你应该注意不要显式指定主键值。关于这个细微差别的更多信息，参见上文的显示指定主键的值 和下文的强制使用`INSERT`或`UPDATE`。

在Django 1.5 和更早的版本中，在设置主键的值时，Django 会作一个`SELECT`。如果`SELECT`找到一行，那么Django 执行`UPDATE`，否则执行`INSERT`。旧的算法导致`UPDATE`情况下多一次查询。有极少数的情况，数据库不会报告有一行被更新，即使数据库包含该对象的主键值。有个例子是PostgreSQL 的`ON UPDATE`触发器，它返回NULL。在这些情况下，可能要通过将`select_on_save`选项设置为`True`以启用旧的算法。

强制使用INSERT 或UPDATE

在一些很少见的场景中，需要强制`save()`方法执行SQL 的`INSERT`而不能执行`UPDATE`。或者相反：更新一行而不是插入一个新行。在这些情况下，你可以传递`force_insert=True`或`force_update=True`参数给`save()`方法。显然，两个参数都传递是错误的：你不可能同时插入和更新！

你应该极少需要使用这些参数。Django 几乎始终会完成正确的事情，覆盖它将导致错误难以跟踪。这个功能只用于高级用法。

使用`update_fields`将强制使用类似`force_update`的更新操作。

基于已存在字段值的属性更新

有时候你需要在一个字段上执行简单的算法操作，例如增加或者减少当前值。实现这点的简单方法是像下面这样：

```
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese
')
>>> product.number_sold += 1
>>> product.save()
```

如果从数据库中读取的旧的 `number_sold` 值为10，那么写回到数据库中的值将为11。

通过将更新基于原始字段的值而不是显式赋予一个新值，这个过程可以避免竞态条件而且更快。Django 提供 `F 表达式` 用于这种类型的相对更新。利用 `F 表达式`，前面的示例可以表示成：

```
>>> from django.db.models import F
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese
')
>>> product.number_sold = F('number_sold') + 1
>>> product.save()
```

更多细节，请参见 `F 表达式` 和它们在更新查询中的用法。

指定要保存的字段

如果传递给 `save()` 的 `update_fields` 关键字参数一个字段名称列表，那么将只有该列表中的字段会被更新。如果你想更新对象的一个或几个字段，这可能是你想要的。不让模型的所有字段都更新将会带来一些轻微的性能提升。例如：

```
product.name = 'Name changed again'
product.save(update_fields=['name'])
```

`update_fields` 参数可以是任何包含字符串的可迭代对象。空的 `update_fields` 可迭代对象将会忽略保存。如果为 `None` 值，将执行所有字段上的更新。

指定 `update_fields` 将强制使用更新操作。

当保存通过延迟模型加载（`only()` 或 `defer()`）进行访问的模型时，只有从数据库中加载的字段才会得到更新。这种情况下，有个自动的 `update_fields`。如果你赋值或者改变延迟字段的值，该字段将会添加到更新的字段中。

删除对象

`Model.delete([using=DEFAULT_DB_ALIAS])`

发出一个SQL `DELETE` 操作。它只在数据库中删除这个对象；其Python 实例仍将存在并持有各个字段的数据。

更多细节，包括如何批量删除对象，请参见删除对象。

如果你想自定义删除的行为，你可以覆盖 `delete()` 方法。详见覆盖预定义的模型方法。

Pickling 对象

当你 `pickle` 一个模型时，它的当前状态是pickled。当你unpickle 它时，它将包含pickle 时模型的实例，而不是数据库中的当前数据。

> 你不可以在不同版本之间共享pickles
>
> 模型的Pickles 只对于产生它们的Django 版本有效。如果你使用Django 版本N pickle，不能保证Django 版本N+1 可以读取这个pickle。Pickles 不应该作为长期的归档策略。
>
> `New in Django 1.8.`
>
> 因为pickle 兼容性的错误很难诊断例如一个悄无声息损坏的对象，当你unpickle 模型使用的Django 版本与pickle 时的不同将引发一个 `RuntimeWarning` 。

其它的模型实例方法

有几个实例方法具有特殊的目的。

> 注
>
> 在Python 3 上，因为所有的字段都原生被认为是Unicode，只需使用 `__str__()` 方法（ `__unicode__()` 方法被废弃）。如果你想与Python 2 兼容，你可以使用 `python_2_unicode_compatible()` 装饰你的模型类。

__unicode__

`Model.__unicode__()`

`__unicode__()` 方法在每当你对一个对象调用unicode() 时调用。Django 在许多地方都使用 `unicode(obj)` （或者相关的函数 `str(obj)` ）。最明显的是在Django 的Admin 站点显示一个对象和在模板中插入对象的值的时候。所以，你应该始终让 `__unicode__()` 方法返回模型的一个友好的、人类可读的形式。

例如：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

如果你定义了模型的 `__unicode__()` 方法且没有定义 `__str__()` 方法，Django 将自动提供一个 `__str__()` ，它调用 `__unicode__()` 并转换结果为一个UTF-8 编码的字符串。下面是一个建议的开发实践：只定义 `__unicode__()` 并让Django 在需要时负责字符串的转换。

__str__

`Model.__str__()`

`__str__()` 方法在每当你对一个对象调用 `str()` 时调用。在Python 3 中，Django 在许多地方使用 `str(obj)` 。最明显的是在Django 的Admin 站点显示一个对象和在模板中插入对象的值的时候。所以，你应该始终让 `__str__()` 方法返回模型的一个友好的、人类可读的形式。

例如：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

在Python 2 中，Django 内部对 `__str__` 的直接使用主要在随处可见的模型的 `repr()` 输出中（例如，调试时的输出）。如果已经有合适的 `__unicode__()` 方法就不需要 `__str__()` 了。

前面 `__unicode__()` 的示例可以使用 `__str__()` 这样类似地编写：

```
from django.db import models
from django.utils.encoding import force_bytes

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        # Note use of django.utils.encoding.force_bytes() here b
ecause
        # first_name and last_name will be unicode strings.
        return force_bytes('%s %s' % (self.first_name, self.last
_name))
```

__eq__

`Model.__eq__()`

定义这个方法是为了让具有相同主键的相同实类的实例是相等的。对于代理模型，实类是模型第一个非代理父类；对于其它模型，它的实类就是模型类自己。

例如：

```
from django.db import models

class MyModel(models.Model):
    id = models.AutoField(primary_key=True)

class MyProxyModel(MyModel):
    class Meta:
        proxy = True

class MultitableInherited(MyModel):
    pass

MyModel(id=1) == MyModel(id=1)
MyModel(id=1) == MyProxyModel(id=1)
MyModel(id=1) != MultitableInherited(id=1)
MyModel(id=1) != MyModel(id=2)
```

> Changed in Django 1.7:
>
> 在之前的版本中，只有类和主键都完全相同的实例才是相等的。

__hash__

`Model.__hash__()`

`__hash__` 方法基于实例主键的值。它等同于hash(obj.pk)。如果实例的主键还没有值，将引发一个 `TypeError` （否则， `__hash__` 方法在实例保存的前后将返回不同的值，而改变一个实例的 `__hash__` 值在Python 中是禁止的）。

```
Changed in Django 1.7:

在之前的版本中，主键没有值的实例是可以哈希的。
```

get_absolute_url

`Model.get_absolute_url()`

`get_absolute_url()` 方法告诉Django 如何计算对象的标准URL。对于调用者，该方法返回的字符串应该可以通过HTTP 引用到这个对象。

例如：

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

（虽然这段代码正确又简单，这并不是编写这个方法可移植性最好的方式。通常使用 `reverse()` 函数是最好的方式。）

例如：

```
def get_absolute_url(self):
    from django.core.urlresolvers import reverse
    return reverse('people.views.details', args=[str(self.id)])
```

Django 使用 `get_absolute_url()` 的一个地方是在Admin 应用中。如果对象定义该方法，对象编辑页面将具有一个“View on site”链接，可以将你直接导入由 `get_absolute_url()` 提供的对象公开视图。

类似地，Django 的另外一些小功能，例如syndication feed 框架 也使用 `get_absolute_url()` 。如果模型的每个实例都具有一个唯一的URL 是合理的，你应该定义 `get_absolute_url()` 。

> 警告
>
> 你应该避免从没有验证过的用户输入构建URL，以减少有害的链接和重定向：

```
def get_absolute_url(self):
    return '/%s/' % self.name
```

> 如果 `self.name` 为'`/example.com`'，将返回'`//example.com/`'， 而它是一个合法的相对URL而不是期望的'`/%2Fexample.com/`'。

在模板中使用 `get_absolute_url()` 而不是硬编码对象的URL 是很好的实践。例如，下面的模板代码很糟糕：

```
<!-- BAD template code. Avoid! -->
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

下面的模板代码要好多了：

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

如果你改变了对象的URL 结构，即使是一些简单的拼写错误，你不需要检查每个可能创建该URL 的地方。在 `get_absolute_url()` 中定义一次，然后在其它代码调用它。

> 注
>
> `get_absolute_url()` 返回的字符串必须只包含ASCII 字符（URI 规范RFC 2396 的要求），并且如需要必须要URL-encoded。
>
> 代码和模板中对 `get_absolute_url()` 的调用应该可以直接使用而不用做进一步处理。你可能想使用 `django.utils.encoding.iri_to_uri()` 函数来帮助你解决这个问题，如果你正在使用ASCII 范围之外的Unicode 字符串。

额外的实例方法

除了 `save()` 、 `delete()` 之外，模型的对象还可能具有以下一些方法：

`Model.get_FOO_display()`

对于每个具有 `choices` 的字段，每个对象将具有一个 `get_FOO_display()` 方法，其中 `FOO` 为该字段的名称。这个方法返回该字段对“人类可读”的值。

例如：

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=2, choices=SHIRT_SI
ZES)
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

`Model.get_next_by_FOO(**kwargs)`

`Model.get_previous_by_FOO(**kwargs)`

如果 `DateField` 和 `DateTimeField` 没有设置 `null=True` ，那么该对象将具有 `get_next_by_FOO()` 和 `get_previous_by_FOO()` 方法，其中 `FOO` 为字段的名称。它根据日期字段返回下一个和上一个对象，并适时引发一个 `DoesNotExist` 。

这两个方法都将使用模型默认的管理器来执行查询。如果你需要使用自定义的管理器或者你需要自定义的筛选，这个两个方法还接受可选的参数，它们应该用字段查询 中提到的格式。

注意，对于完全相同的日期，这些方法还将利用主键来进行查找。这保证不会有记录遗漏或重复。这还意味着你不可以在未保存的对象上使用这些方法。

其它属性

DoesNotExist

exception `Model.DoesNotExist`

ORM 在好几个地方会引发这个异常，例如 `QuerySet.get()` 根据给定的查询参数找不到对象时。

Django 为每个类提供一个 `DoesNotExist` 异常属性是为了区别找不到的对象所属的类，并让你可以利用 `try/except` 捕获一个特定模型的类。这个异常是 `django.core.exceptions.ObjectDoesNotExist` 的子类。

> 译者：Django 文档协作翻译小组，原文：Instance methods。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

关联对象参考

class RelatedManager

"关联管理器"是在一对多或者多对多的关联上下文中使用的管理器。它存在于下面两种情况：

ForeignKey关系的“另一边”。像这样：

```
from django.db import models

class Reporter(models.Model):
    # ...
    pass

class Article(models.Model):
    reporter = models.ForeignKey(Reporter)
```

在上面的例子中，管理器reporter.article_set拥有下面的方法。

ManyToManyField关系的两边：

```
class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    toppings = models.ManyToManyField(Topping)
```

这个例子中，topping.pizza_set 和pizza.toppings都拥有下面的方法。

add(obj1[, obj2, ...])

把指定的模型对象添加到关联对象集中。

例如：

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associates Entry e with Blog b.
```

在上面的例子中，对于ForeignKey关系，e.save()由关联管理器调用，执行更新操作。然而，在多对多关系中使用add()并不会调用任何 save()方法，而是由QuerySet.bulk_create()创建关系。如果你需要在关系被创建时执行一些自定义的逻辑，请监听m2m_changed信号。

create(**kwargs)

创建一个新的对象，保存对象，并将它添加到关联对象集之中。返回新创建的对象：

```
>>> b = Blog.objects.get(id=1)
>>> e = b.entry_set.create(
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )

# No need to call e.save() at this point -- it's already been sa
ved.
```

这完全等价于（不过更加简洁于）：

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry(
...     blog=b,
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )
>>> e.save(force_insert=True)
```

要注意我们并不需要指定模型中用于定义关系的关键词参数。在上面的例子中，我们并没有传入blog参数给create()。Django会明白新的 Entry对象blog 应该添加到b中。

remove(obj1[, obj2, ...])

从关联对象集中移除执行的模型对象：

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) # Disassociates Entry e from Blog b.
```

和add()相似，上面的例子中，e.save()可会执行更新操作。但是，多对多关系上的remove()，会使用QuerySet.delete()删除关系，意思是并不会有任何模型调用save()方法：如果你想在一个关系被删除时执行自定义的代码，请监听m2m_changed信号。

对于ForeignKey对象，这个方法仅在null=True时存在。如果关联的字段不能设置为None (NULL)，则这个对象在添加到另一个关联之前不能移除关联。在上面的例子中，从b.entry_set()移除e等价于让e.blog = None，由于blog的ForeignKey没有设置null=True，这个操作是无效的。

对于ForeignKey对象，该方法接受一个bulk参数来控制它如果执行操作。如果为True（默认值），QuerySet.update()会被使用。而如果bulk=False，会在每个单独的模型实例上调用save()方法。这会触发pre_save和post_save，它们会消耗一定的性能。

clear()

从关联对象集中移除一切对象。

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.clear()
```

注意这样不会删除对象 —— 只会删除他们之间的关联。

就像 remove() 方法一样，clear()只能在 null=True的ForeignKey上被调用，也可以接受bulk关键词参数。

> 注意
>
> 注意对于所有类型的关联字段，add()、create()、remove()和clear()都会马上更新数据库。换句话说，在关联的任何一端，都不需要再调用save()方法。
>
> 同样，如果你再多对多关系中使用了中间模型，一些关联管理的方法会被禁用。

直接赋值

通过赋值一个新的可迭代的对象，关联对象集可以被整体替换掉。

```
>>> new_list = [obj1, obj2, obj3]
>>> e.related_set = new_list
```

如果外键关系满足null=True，关联管理器会在添加new_list中的内容之前，首先调用clear()方法来解除关联集中一切已存在对象的关联。否则， new_list中的对象会在已存在的关联的基础上被添加。

迁移

迁移

New in Django 1.7.

迁移是Django用于同步你的发生改变的模型(添加一个字段，删除一个模型，等等。) 到你的数据库。 它的设计是很智能的, 但是你还是需要了解什么时候进行迁移, 什么时候去启动它们, 以及可能遇到的指令问题。

简短历史

在 1.7版本之前, Django 只支持添加新模型到数据库；无法通过 `syncdb` 命令来修改或移除已存在的模型 (已被 `migrate` 代替)。

第三方工具，最著名的是South, 为这些额外的功能提供支持,但是它还是被认为是很重要的部分并且加入到django的核心里面。

命令集

有一些你用于进行迁移的命令集和数据库架构的django操作

- `migrate` , 负责执行迁移, 以及撤销和列出迁移的状态。
- `makemigrations` , 负责基于你的模型修改创建一个新的迁移
- `sqlmigrate` , 展示迁移的sql语句

值得注意的是，迁移是建立和运行在每一个app的基础上。 特别的,确实存在 没有用迁移的*app* (它们被称为“非迁移”apps) -相比利用遗留的保存行为，它们只是在这个基础上添加一个新的模型.

你可以想象 migrations相当一个你的数据库的一个版本控制系统。 `makemigrations` 命令负责保存你的模型变化到一个迁移文件 - 和 commits 很类似 - 同时 `migrate` 负责将改变提交到数据库。

每个app 的迁移文件会保存到每个相应app的“migrations”文件夹里面,并且准备如何去执行它, 作为一个分布式代码库。 每当在你的开发机器或是你同事的机器并且最终在你的生产机器上运行同样的迁移，你应当再创建这些文件。

注意

通过修改 `MIGRATION_MODULES` 设置，可以覆盖那些每个app 都包含 migrations 的 package

同样的方式，同样的数据集，将产生一致的结果, 那意味着你在开发和筹划中在按照同样的原理运行的情况下得到的结果和线上是一致的

Django 将迁移你对模型和字段做出的任何改变 - 甚至包括那些对数据库没有影响的操作， -在历史记录存放所有改变是正确重建field的唯一方式，你可能会在以后的数据迁移中使用到这些选项 （比如，在你定制了校验器的时候）。

后台支持

Django附带的所有后端都支持迁移，以及任何第三方后端，如果他们已编程支持模式更改（通过*SchemaEditor*类完成）。

但是，一些数据库比其他数据库在模式迁移方面更有能力；下面覆盖的一些警告.

PostgreSQL

PostgreSQL是所有数据库中最能够支持的模式；唯一的警告是添加具有默认值的列将导致表的完全重写，其时间与其大小成比例。

因此，建议您始终使用 `null=True` 创建新列，因为这样会立即添加。

MySQL

MySQL缺少对模式更改操作的事务的支持，这意味着如果迁移无法应用，则必须手动取消批准更改才能再次尝试（不可能回滚到更早的点）。

此外，MySQL将为每个模式操作完全重写表，通常需要与表中的行数成正比的时间来添加或删除列。在较慢的硬件上，这可能比每分钟每百万行更糟 - 向表中添加几列只有几百万行可能会锁定您的网站超过十分钟。

最后，MySQL对列，表和索引的名称长度有相当小的限制，以及索引涵盖的所有列的组合大小的限制。这意味着在其他后端可能的索引将无法在MySQL下创建。

SQLite

SQLite具有非常少的内置模式更改支持，因此Django尝试通过以下方式模拟它：

- 使用新模式创建新表
- 复制数据
- 删除旧表
- 重命名新表以匹配原始名称

这个过程一般效果很好，但它可能很慢，偶尔会出现问题。不建议您在生产环境中运行和迁移SQLite，除非您非常了解风险及其限制；支持Django的设计允许开发人员在其本地机器上使用SQLite开发不太复杂的Django项目，而不需要一个完整的数据库。

工作流程

迁移工作很简单。 修改你的模型 -比如添加字段和移除一个模型 - 然后运行 `makemigrations` :

```
$ python manage.py makemigrations
Migrations for 'books':
  0003_auto.py:
    - Alter field author on book
```

你的原型会扫描和比较你当前迁移文件里面的版本,同时新的迁移文件会被创建. 请务必查阅输出，看看 `makemigrations` 是如何理解你做出的更改 - 迁移不是完全准确的, 对于一些复杂的更改，它可能不会检测到你所期望的东西。

一旦你有了新的迁移文件， 你应该把它们提交到你的数据库以确保它们像预期的那样工作：

```
$ python manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: sessions, admin, messages, auth,
staticfiles, contenttypes
  Apply all migrations: books
Synchronizing apps without migrations:
  Creating tables...
  Installing custom SQL...
  Installing indexes...
Installed 0 object(s) from 0 fixture(s)
Running migrations:
  Applying books.0003_auto... OK
```

命令分两步运行，首先，它会同步未迁移的应用（与旧的 `syncdb` 命令所做的操作相同），然后对未应用的更改进行迁移。

一旦迁移应用后，在版本控制系统中将迁移与模型的改变作为同一次提交-那样的话，其他开发者（或者你的生产服务器）检查代码时，他们将同时看到模型的改变及伴随的迁移。

New in Django 1.8.

If you want to give the migration(s) a meaningful name instead of a generated one, you can use the `--name` option:使用--name 选项自定义migrations迁移文件名称。

```
$ python manage.py makemigrations --name changed_my_model your_a
pp_label
```

版本控制

由于迁移存储在版本控制中，因此您偶尔会遇到这样的情况：您和另一个开发人员同时提交了迁移到同一个应用程序，导致两个迁移具有相同的编号。

不要担心 - 数字只是为了开发人员的参考，Django只关心每个迁移有不同的名称。迁移指定文件中依赖的其他迁移（包括在同一应用中的早期迁移），因此可以检测同一应用的两次新迁移是否未订购。

当发生这种情况时，Django会提示你并给你一些选项。如果它认为它足够安全，它会提供自动线性化两个迁移为你。如果没有，您必须自行修改迁移作业，别担心，这并不难，在下面的*Migration files*中会有更多说明。

依赖关系

虽然迁移是基于应用程序的，但是模型所包含的表和关系太复杂，无法一次为一个应用程序创建。当您进行需要其他方式运行的迁移时，例如，您可以在 `books` 应用程序中将 `ForeignKey` 添加到您的 `authors` 导致的迁移将包含对 `authors` 中的迁移的依赖。

这意味着当您运行迁移时， `authors` 迁移首先运行，创建 `ForeignKey` 引用的表，然后进行 `ForeignKey` 如果没有发生这种情况，迁移将尝试创建 `ForeignKey` 列，而不引用现有的表，并且您的数据库会抛出错误。

此依赖关系行为会影响大多数将操作限制为单个应用程序的迁移操作。限制到单个应用程序（在 `makemigrations` 或 `migrate` ）是一个尽力而为的承诺，而不是保证；任何其他需要用来获取依赖关系的应用程序都会正确。

不过请注意，由于没有迁移，未迁移的应用程式无法依赖已迁移的应用程式。这意味着通常不可能有一个未迁移的应用程序具有 `ForeignKey` 或 `ManyToManyField` 到已迁移的应用程序；一些情况可能会工作，但它最终会失败。

警告

即使事情似乎与未迁移的应用程序一起工作，这取决于已迁移的应用程序，Django可能不会生成所有必需的外键约束！

如果您使用可交换模型（例如 `AUTH_USER_MODEL` ），这一点尤其明显，因为使用可交换模型的每个应用都需要进行迁移（如果您不幸）。随着时间的推移，越来越多的第三方应用将获得迁移，但在此期间，您可以自行迁移（如果您愿意，可以使用 `MIGRATION_MODULES` 将这些模块存储在应用自己的模块之外））,或保持应用程序与您的用户模型未迁移。

迁移文件

迁移存储为磁盘格式，此处称为“迁移文件”。这些文件实际上只是正常的Python文件，具有约定的对象布局，以声明样式编写。

基本迁移文件如下所示：

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [("migrations", "0001_initial")]

    operations = [
        migrations.DeleteModel("Tribble"),
        migrations.AddField("Author", "rating", models.IntegerFi
eld(default=0)),
    ]
```

加载迁移文件（作为Python模块）时，Django查找的是 `django.db.migrations.Migration` 的子类，名为 `Migration` 。然后它检查这个对象的四个属性，大多数时间只使用其中的两个：

- `dependencies` ，此依赖关系的迁移列表。
- `operations` ，定义此迁移操作的 `Operation` 类的列表。

operations 是关键，它们是一个陈述性说明的集合，能告诉Django需要产生什么样的schema 更改。Django扫描它们并构建所有应用程序的所有模式更改的内存中表示，并使用它来生成使模式更改的SQL。

内存结构也用于计算模型和迁移的当前状态之间的差异；Django运行所有的更改，按顺序，在内存中的模型集，以找出你的模型的上次你运行 `makemigrations` 的状态。然后使用这些模型与您的 `models.py` 文件中的模型进行比较，以确定您所做的更改。

您应该很少（如果曾经）需要手动编辑迁移文件，但是如果需要，完全可以手动编写它们。一些更复杂的操作不是自动检测的，只能通过手写迁移提供，所以如果你必须编辑它们，不要害怕。

自定义字段

您无法修改已迁移的自定义字段中的位置参数的数量，而不提高 `TypeError` 。旧迁移将调用具有旧签名的修改的 `__init__` 方法。So if you need a new argument, please create a keyword argument and add something like `assert 'argument_name' in kwargs` in the constructor.

模型管理者

New in Django 1.8.

您可以选择将管理器序列化为迁移，并使它们在 `RunPython` 操作中可用。这是通过在管理器类上定义 `use_in_migrations` 属性来实现的：

```
class MyManager(models.Manager):
    use_in_migrations = True

class MyModel(models.Model):
    objects = MyManager()
```

如果您使用 `from_queryset()` 函数动态生成管理器类，则需要从生成的类继承以使其可导入：

```
class MyManager(MyBaseManager.from_queryset(CustomQuerySet)):
    use_in_migrations = True

class MyModel(models.Model):
    objects = MyManager()
```

请参阅迁移中有关历史模型的注意事项，了解其中的影响。

将迁移添加到应用

将迁移添加到新应用程序很简单 - 它们已预先配置为接受迁移，因此，一旦进行了某些更改，只需运行 `makemigrations` 即可。

如果您的应用程序已经有模型和数据库表，但还没有迁移（例如，您是根据以前的Django版本创建的），则需要将其转换为使用迁移；这是一个简单的过程：

```
$ python manage.py makemigrations your_app_label
```

这将为您的应用程序进行新的初始迁移。现在，执行 `python manage.py migrate - fake-initial` Django将检测到您有一个初始迁移（没有 `--fake-initial` 标志，`migrate` 命令将错误输出，因为它要创建的表已经存在。

注意，这只工作给予两件事：

- 你没有改变你的模型，因为你制作了他们的表。要使迁移正常工作，必须先进行初始迁移，然后进行更改，因为Django会将更改与迁移文件进行比较，而不是数据库。
- 您没有手动编辑数据库 - Django将无法检测到您的数据库与您的模型不匹配，您只会在迁移尝试修改这些表时收到错误。

历史模型

当您运行迁移时，Django使用存储在迁移文件中的模型的历史版本。如果您使用 RunPython 操作编写Python代码，或者如果在数据库路由器上使用 allow_migrate 方法，则会暴露给这些版本的模型。

因为不可能序列化任意Python代码，所以这些历史模型不会有你定义的任何自定义方法。但是，他们将具有相同的字段，关系，管理员（仅限于 use_in_migrations = True ）和 Meta 选项（也已版本化，因此它们可能与您当前的不同）。

警告

这意味着，当您在迁移中访问对象时，您不会有对对象调用的自定义 save() 方法，并且您不会有任何自定义构造函数或实例方法。计划适当！

对诸如 upload_to 和 limit_choices_to 等字段选项中的函数的引用以及具有 use_in_migrations = t6 > True 在迁移中序列化，因此只要存在引用它们的迁移，函数和类就需要保留。还需要保留任何*custom model fields*，因为这些是通过迁移直接导入的。

此外，模型的基类只存储为指针，因此只要存在包含对它们的引用的迁移，就必须始终保持基类。在正面，这些基类的方法和管理器通常继承，所以如果你绝对需要访问这些，你可以选择将它们移动到超类。

删除模型字段时的注意事项

New in Django 1.8.

与上一节中介绍的“对历史函数的引用”注意事项类似，如果在旧迁移中引用了自定义模型字段，则从项目或第三方应用中移除自定义模型字段将会导致问题。

为了帮助解决这种情况，Django提供了一些模型字段属性，以帮助使用*system checks framework*来取消模型字段。

将 system_check_deprecated_details 属性添加到模型字段，类似于以下内容：

```
class IPAddressField(Field):
    system_check_deprecated_details = {
        'msg': (
            'IPAddressField has been deprecated. Support for it
(except '
            'in historical migrations) will be removed in Django
 1.9.'
        ),
        'hint': 'Use GenericIPAddressField instead.',  # optiona
l
        'id': 'fields.W900',  # pick a unique ID for your field.
    }
```

在您选择的弃用期之后（Django本身的字段的两个主要版本），将 `system_check_deprecated_details` 属性更改为 `system_check_removed_details` 并更新字典类似于：

```
class IPAddressField(Field):
    system_check_removed_details = {
        'msg': (
            'IPAddressField has been removed except for support
in '
            'historical migrations.'
        ),
        'hint': 'Use GenericIPAddressField instead.',
        'id': 'fields.E900',  # pick a unique ID for your field.
    }
```

您应该保留字段的数据库迁移操作所需的方法，例如 `__init__()`， `deconstruct()` 和 `get_internal_type()` 只要引用此字段的任何迁移存在，就保留此存根字段。例如，在压缩迁移并删除旧迁移后，您应该可以完全删除该字段。

数据迁移

除了更改数据库模式之外，您还可以使用迁移来更改数据库中的数据，如果需要，还可以与模式一起使用。

更改数据的迁移通常称为"数据迁移"；他们最好写成单独的迁移，坐在模式迁移。

Django不能为你自动生成数据迁移，就像它对模式迁移一样，但是写它们并不困难。Django中的迁移文件由*Operations*组成，用于数据迁移的主要操作是 `RunPython` 。

首先，让一个空的迁移文件，你可以工作（Django会把文件放在正确的地方，建议一个名字，并为你添加依赖关系）：

```
python manage.py makemigrations --empty yourappname
```

接下来, 我们打开刚刚生成的迁移文件; 你应该可以看到类似以下的内容:

```
# -*- coding: utf-8 -*-
from django.db import models, migrations

class Migration(migrations.Migration):

    dependencies = [
        ('yourappname', '0001_initial'),
    ]

    operations = [
    ]
```

现在，所有你需要做的是创建一个新函数，并使用 `RunPython` 。 `RunPython` 需要一个可调用作为其参数，它需要两个参数 - 第一个是一个*app registry*，它将所有模型的历史版本加载到其中以匹配历史记录迁移就位，第二个是*SchemaEditor*，您可以使用它手动实现数据库模式更改（但要小心，这样做可能会使迁移自动检测器混乱）。

让我们编写一个简单的迁移，用 `first_name` 和 `last_name` 的组合值填充新的 `name` 字段（我们来到了我们的感觉不是每个人都有名和姓）。所有我们需要做的是使用历史模型并在行上进行迭代：

```
# -*- coding: utf-8 -*-
from django.db import models, migrations

def combine_names(apps, schema_editor):
    # We can't import the Person model directly as it may be a n
ewer
    # version than this migration expects. We use the historical
 version.
    Person = apps.get_model("yourappname", "Person")
    for person in Person.objects.all():
        person.name = "%s  %s" % (person.first_name, person.last
_name)
        person.save()

class Migration(migrations.Migration):

    dependencies = [
        ('yourappname', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(combine_names),
    ]
```

完成后，我们可以正常运行 `python manage.py migrate` 与其他迁移一起放置。

您可以传递第二个可调用项到 `RunPython` ，以便在向后迁移时运行您希望执行的任何逻辑。如果省略此可调用项，向后迁移将引发异常。

通过其他应用访问模型

When writing a `RunPython` function that uses models from apps other than the one in which the migration is located, the migration's `dependencies` attribute should include the latest migration of each app that is involved, otherwise you may get an error similar to: `LookupError: No installed app with label 'myappname'` when you try to retrieve the model in the `RunPython` function using `apps.get_model()` .

在下面的示例中，我们在 `app1` 中有一个迁移，需要使用 `app2` 中的模型。我们不关心 `move_m1` 的细节，而是需要从两个应用程序访问模型的事实。因此，我们添加了一个依赖关系，指定 `app2` 的最后一次迁移：

```
class Migration(migrations.Migration):

    dependencies = [
        ('app1', '0001_initial'),
        # added dependency to enable using models from app2 in m
ove_m1
        ('app2', '0004_foobar'),
    ]

    operations = [
        migrations.RunPython(move_m1),
    ]
```

更多高级迁移

如果你对更高级的迁移操作感兴趣，或者想写自定义迁移文件, see the *migration operations reference* and the “how-to” on *writing migrations*.

挤压迁移

鼓励你自由迁徙，不要担心你有多少；迁移代码被优化以一次处理几百个而没有大的减速。然而，最终你会想从几百个迁移回到几个，这就是挤压的地方。

压缩是将现有的许多迁移集减少到仍然表示相同更改的一个（或有时几个）迁移的行为。

Django通过采取所有现有迁移，提取其 `Operation` 并将它们全部顺序，然后在它们上运行优化器来尝试并减少列表的长度，这样做 - 例如，它知道 `CreateModel` 和 `DeleteModel` 互相取消，并且知道 `AddField` 可以滚动到 `CreateModel` 。

一旦操作序列被尽可能地减少 - 可能的数量取决于模型是多么密切，如果你有任何 `RunSQL` 或 `RunPython` 操作（它可以'通过优化） - Django然后将其写回一组新的初始迁移文件。

这些文件被标记为说它们替换以前被压缩的迁移，因此它们可以与旧的迁移文件共存，并且Django将根据您在历史记录中的位置智能地在它们之间切换。如果您仍然部分地通过您压缩的迁移集，它将继续使用它们，直到它到达结束，然后切换到压缩的历史记录，而新的安装将只使用新的压缩迁移，并跳过所有的旧那些。

这使您可以压缩，而不是混乱当前生产中尚未完全更新的系统。建议的过程是压缩，保留旧文件，提交和释放，等待所有系统升级到新版本（或如果您是第三方项目，只是确保您的用户升级版本，而不跳过任何），然后删除旧的文件，提交并做第二个版本。

支持所有这一切的命令是 `squashmigrations` - 只是将它的应用程序标签和迁移名称，你想要压缩，它会工作：

```
$ ./manage.py squashmigrations myapp 0004
Will squash the following migrations:
 - 0001_initial
 - 0002_some_change
 - 0003_another_change
 - 0004_undo_something
Do you wish to proceed? [yN] y
Optimizing...
  Optimized from 12 operations to 7 operations.
Created new squashed migration /home/andrew/Programs/DjangoTest/
test/migrations/0001_squashed_0004_undo_somthing.py
  You should commit this migration but leave the old ones in pla
ce;
  the new migration will be used for new installs. Once you are
sure
  all instances of the codebase have applied the migrations you
squashed,
  you can delete them.
```

注意，Django中的模型相互依赖性可能变得非常复杂，压缩可能导致不运行的迁移；（在这种情况下，您可以使用 `--no-optimize` 再试一次，虽然您也应该报告问题），或者使用 `CircularDependencyError` 您可以手动解决它。

要手动解析 `CircularDependencyError` ，请将循环依赖关系循环中的一个ForeignKeys拆分为单独的迁移，并使用它移动其他应用程序的依赖关系。如果您不确定，请问当您从模型中创建全新的迁移时，makemigrations如何处理这个问题。在将来的Django版本中，将会更新squashmigrations以尝试自己解决这些错误。

一旦您压缩了迁移，您应该将其替换为它替换的迁移，并将此更改分发到应用程序的所有正在运行的实例，确保它们运行 `migrate` 以将更改存储在其数据库中。

完成此操作后，您必须通过以下方式将压缩的迁移转换为正常的初始迁移：

- 删除它替换的所有迁移文件
- 删除缩小迁移的 `Migration` 类中的 `replaces` 参数（这是Django如何解释它是一个被压缩的迁移）

注意

一旦您压制了迁移，您就不应该重新压缩被压缩的迁移，直到您完全转换为正常迁移为止。

序列化值

迁移只是包含模型的旧定义的Python文件 - 因此，为了编写它们，Django必须获取模型的当前状态并将它们序列化到一个文件中。

虽然Django可以序列化大多数东西，但有一些事情，我们不能序列化为有效的Python表示 - 没有Python标准如何将值转换回代码（ `repr()` 仅适用于基本值，并且不指定导入路径）。

Django可以序列化以下内容：

- `int` , `long` , `float` , `bool` , `str` , `unicode` , `bytes` , `None`
- `list` ， `set` ， `tuple` ， `dict`
- `datetime.date` ， `datetime.time` 和 `datetime.datetime` 实例（包括时区感知的实例）
- `decimal.` 小数实例
- 任何Django字段
- 任何函数或方法引用（例如 `datetime.datetime.today` ）（必须在模块的顶级作用域中）
- 任何类引用（必须在模块的顶级作用域中）
- 使用自定义 `deconstruct()` 方法（*see below*）的任何内容

Changed in Django 1.7.1:

支持串行化时区感知的数据时间。

Django只能在Python 3上序列化以下内容：

- 在类体内使用的未绑定方法（见下文）

Django不能序列化：

- 嵌套类
- 任意类实例（例如 `MyClass（4.3， 5.7）` ）
- Lambdas

由于 `__qualname__` 仅在Python 3中引入，Django只能序列化Python 3上的以下模式（在类体中使用的未绑定方法），并且将无法序列化对它的引用Python 2：

```
class MyModel(models.Model):

    def upload_to(self):
        return "something dynamic"

    my_file = models.FileField(upload_to=upload_to)
```

如果您使用的是Python 2，我们建议您移动upload_to的方法以及接受可调用项（例如 `default` ）的类似参数，使其位于主模块正文中，而不是类主体中。

添加deconstruct()方法

您可以让Django通过为类提供 `deconstruct()` 方法来序列化您自己的自定义类实例。它不需要参数，应该返回一个三元组的元组 `（path， args， kwargs）`

- `path` 应该是类的Python路径，类名包含在最后一部分中（例如 `myapp.custom_things.` 我的课）。如果你的类在模块的顶层不可用，那么它是不可序列化的。
- `args` 应该是要传递给类' `__init__` 方法的位置参数列表。这个列表中的所有内容都应该是可序列化的。
- `kwargs` 应该是传递到类' `__init__` 方法的关键字参数的dict。每个值本身都是可序列化的。

注意

此返回值不同于自定义字段的 `deconstruct()` 方法*for custom fields*

Django将写出这个值作为你的类的实例化与给定的参数，类似于它写出对Django字段的引用的方式。

为了防止每次运行 `makemigrations` 时创建新的迁移，您还应该向装饰类添加一个 `__eq__()` 方法。这个函数将被Django的迁移框架调用来检测状态之间的变化。

只要你的类的构造函数的所有参数都是可序列化的，你可以使用 `django.utils.deconstruct` 中的 `@deconstructible` 类装饰器来添加 `deconstruct()` 方法：

```
from django.utils.deconstruct import deconstructible

@deconstructible
class MyCustomClass(object):

    def __init__(self, foo=1):
        self.foo = foo
        ...

    def __eq__(self, other):
        return self.foo == other.foo
```

装饰器添加逻辑来捕获和保存参数到它们的构造函数中，然后在调用deconstruct()时返回这些参数。

Python 2 和 3 支持

为了生成支持Python 2和3的迁移，模型和字段中使用的所有字符串文字（例如 `verbose_name` ，`related_name` 等）必须始终为在Python 2和3（而不是Python 2中的字节和Python 3中的文本，未标记的字符串文字的默认情况）下的字符串或文本（unicode）字符串。否则在Python 3下运行 `makemigrations` 会生成虚假的新迁移，将所有这些字符串属性转换为文本。

实现这个的最简单的方法是遵循Django的*Python 3 porting guide*中的建议，并确保所有的模块以 开头 __ future __ t4> import unicode_literals ，以便所有未标记的字符串文字总是unicode，而不管Python版本。当您将它添加到在Python 2上生成的现有迁移的应用程序时，下一次在Python 3上运行 `makemigrations` 可能会生成许多更改，因为它将所有的bytestring属性转换为文本字符串；这是正常的，应该只发生一次。

多Django版本支持

如果您是使用模型的第三方应用的维护者，则可能需要提供支持多个Django版本的迁移。在这种情况下，您应始终使用希望支持的Django版本运行 `makemigrations` 。

迁移系统将根据与其他Django相同的策略保持向后兼容性，因此在Django X.Y上生成的迁移文件应该在Django X.Y + 1上保持不变。然而，迁移系统不承诺向前兼容性。可能会添加新功能，并且使用较新版本的Django生成的迁移文件可能无法在旧版本上运行。

从 South 升级

如果你有使用 South创建的迁移, 使用 `django.db.migrations` 升级的过程非常简单：

- 确认所有的安装都以South创建的迁移完全更新。
- 从 `INSTALLED_APPS` 中移除 `'south'`
- 删除你所有的迁移文件（编号标识的），但不能删除所在目录或 `__init__.py` - 也需确认删除 `.pyc` 文件。
- 运行 `python manage.py makemigrations` . Django应该看到空的迁移目录，并以新的格式进行新的初始迁移。
- 执行 `python manage.py 迁移 - fake-initial` 。Django会看到初始迁移的表已经存在，并将它们标记为已应用，而不运行它们。（Django不会检查表模式是否与您的模型匹配，只是正确的表名存在）。

而已！唯一的复杂因素是如果你有一个循环依赖循环的外键；在这种情况下， `makemigrations` 可能会进行多个初始迁移，您需要将它们全部标记为已应用：

```
python manage.py migrate --fake yourappnamehere
```

Changed in Django 1.8:

`--fake-initial` 标志已添加到 `migrate` ；以前，如果检测到现有表，初始迁移总是自动伪造应用。

库/第三方应用

如果你是一个库或应用程序维护者，并且希望支持南迁移（对于Django 1.6及以下版本）和Django迁移（对于1.7及以上版本），应在应用程序中保留两个并行迁移集，每个格式一个。

为了帮助这一点，South 1.0将首先在 `south_migrations` 目录中自动查找South格式迁移，然后查看 `migrations` ，这意味着用户的项目将透明地使用正确的只要将您的南迁移位于 `south_migrations` 目录中，并将您的Django迁移位于 `migrations` 目录中即可。

更多信息请参考 South 1.0 发布说明.

参考

The Migrations Operations Reference

Covers the schema operations API, special operations, and writing your own operations.

The Writing Migrations “how-to”

Explains how to structure and write database migrations for different scenarios you might encounter.

迁移操作

迁移文件由一个或多个 `Operation` 组成，这些对象声明性地记录迁移应对数据库执行的操作。

Django还使用这些 `Operation` 对象来计算您的模型在历史上的样子，并计算自上次迁移后对模型所做的更改，以便自动编写迁移；这就是为什么他们是声明性的，因为它意味着Django可以轻松地将它们全部加载到内存中，并通过它们运行，而不用触及数据库，以确定您的项目应该是什么样子。

还有更专门的 `Operation` 对象，用于像*data migrations*和高级手动数据库操作。如果要封装常用的自定义更改，也可以编写自己的 `Operation` 类。

如果您需要一个空的迁移文件来写自己的 `Operation` 对象，只需使用 `python manage.py makemigrations - 空的 yourappname` ，但请注意，手动添加模式更改操作可能会混淆迁移自动检测器并导致运行 `makemigrations` 输出不正确的代码。

所有的核心Django操作都可以从 `django.db.migrations.operations` 模块获得。

有关介绍材料，请参阅*migrations topic guide*。

模式操作

CreateModel

class `CreateModel` (*name*, *fields*, *options=None*, *bases=None*, *managers=None*)

在项目历史中创建一个新模型，并在数据库中创建一个相应的表来匹配它。

`name` 是型号名称，如写在 `models.py` 文件中。

`fields` 是 `（field_name， field_instance）` 的2元组的列表。字段实例应为未绑定字段（因此只有 `models.` CharField()，而不是一个字段从另一个模型）。

`options` 是来自模型的 `Meta` 类的值的可选字典。

`bases` 是此模型继承的其他类的可选列表；它可以包含类对象以及格式为 `"appname.ModelName"` 的字符串，如果你想依赖另一个模型（所以你从历史版本继承）。如果没有提供，它默认只继承标准的 `models.Model` 。

`managers` 获取 `（manager_name， manager_instance）` 的2元组列表。列表中的第一个管理器将成为迁移期间此模型的默认管理器。

Changed in Django 1.8:

已添加 `managers` 参数。

删除模型

class `DeleteModel` (*name*)

从数据库中删除项目历史中的模型及其表。

重命名模型

class `RenameModel` (*old_name*, *new_name*)

将模型从旧名称重命名为新名称。

如果您立即更改模型的名称及其几个字段，则可能必须手动添加此字段；到自动检测器，这看起来像您删除了一个具有旧名称的模型，并添加了一个具有不同名称的模型，并且其创建的迁移将丢失旧表中的任何数据。

AlterModelTable

class `AlterModelTable` (*name*, *table*)

更改模型的表名（ `Meta` 子类上的 `db_table` 选项）。

AlterUniqueTogether

class `AlterUniqueTogether` (*name*, *unique_together*)

更改模型的唯一约束集（ `Meta` 子类上的 `unique_together` 选项）。

AlterIndexTogether

class `AlterIndexTogether` (*name*, *index_together*)

更改模型的自定义索引集（ `Meta` 子类上的 `index_together` 选项）。

AlterOrderWithRespectTo

class `AlterOrderWithRespectTo` (*name*, *order_with_respect_to*)

在 `Meta` 子类上为 `order_with_respect_to` 选项创建或删除所需的 `_order` 列。

AlterModelOptions

class `AlterModelOptions` (*name*, *options*)

存储对 `permissions` 和 `verbose_name` 的杂项模型选项（模型 `Meta` 不会影响数据库，但会对 `RunPython` 实例保留这些更改以使用。 `options` 应是将选项名称映射到值的字典。

AlterModelManagers

New in Django 1.8.

class `AlterModelManagers` (*name*, *managers*)

更改迁移期间可用的管理器。

AddField

class `AddField` (*model_name*, *name*, *field*, *preserve_default=True*)

向模型中添加字段。 `model_name` 是模型的名称， `name` 是字段的名称， `field` 是未绑定的Field实例在 `models.py` 中 - 例如， `models.` IntegerField（null = True）。

The `preserve_default` argument indicates whether the field’s default value is permanent and should be baked into the project state (`True`), or if it is temporary and just for this migration (`False`) - usually because the migration is adding a non-nullable field to a table and needs a default value to put into existing rows. 它不影响在数据库中直接设置默认值的行为 - Django从不设置数据库默认值，并始终将它们应用于Django ORM代码。

RemoveField

class `RemoveField` (*model_name*, *name*)

从模型中删除字段。

记住，当颠倒这实际上是添加一个字段到模型；如果字段不可为空，这可能使此操作不可逆（除了任何数据丢失，这当然是不可逆的）。

AlterField

class `AlterField` (*model_name*, *name*, *field*, *preserve_default=True*)

更改字段的定义，包括对其类型， `null` ， `unique` ， `db_column` 和其他字段属性的更改。

The `preserve_default` argument indicates whether the field’s default value is permanent and should be baked into the project state (`True`), or if it is temporary and just for this migration (`False`) - usually because the migration is

altering a nullable field to a non-nullable one and needs a default value to put into existing rows. 它不影响在数据库中直接设置默认值的行为 - Django从不设置数据库默认值，并始终将它们应用于Django ORM代码。

请注意，并非所有数据库都可以进行所有更改 - 例如，您无法更改类似 `models.` TextField()转换为数字类型字段，如 `models.` 大多数数据库上的IntegerField()。

Changed in Django 1.7.1:

已添加 `preserve_default` 参数。

RenameField

class `RenameField` (*model_name*, *old_name*, *new_name*)

更改字段名称（除非设置 `db_column` ，否则其列名称）。

特别行动

RunSQL

class `RunSQL` (*sql*, *reverse_sql=None*, *state_operations=None*, *hints=None*)

允许在数据库上运行任意SQL - 对于Django不能直接支持的数据库后端的更高级功能（如部分索引）非常有用。

`sql` 和 `reverse_sql` （如果提供）应为在数据库上运行的SQL字符串。在大多数数据库后端（除PostgreSQL之外），Django将在执行它们之前将SQL拆分为单独的语句。这需要安装sqlparse Python库。

您还可以传递字符串或2元组的列表。后者用于以与*cursor.execute()*相同的方式传递查询和参数。这三个操作是等效的：

```
migrations.RunSQL("INSERT INTO musician (name) VALUES ('Reinhard
t');")
migrations.RunSQL(["INSERT INTO musician (name) VALUES ('Reinhar
dt');", None])
migrations.RunSQL(["INSERT INTO musician (name) VALUES (%s);", [
'Reinhardt']])
```

如果要在查询中包括文字百分号，则必须将它们加倍（如果您正在传递参数）。

`state_operations` 参数是这样的，您可以根据项目状态提供等同于SQL的操作；例如，如果您手动创建列，则应在此处传递包含 `AddField` 操作的列表，以便自动检测器仍然具有模型的最新状态（否则，当您下一次运

行 `makemigrations` ，它将不会看到任何添加该字段的操作，因此将尝试再次运行它）。

可选的 `hints` 参数将作为 `**hints` 传递到数据库路由器的 `allow_migrate()` 方法，以帮助他们进行路由决策。有关数据库提示的更多详细信息，请参阅*Hints*。

Changed in Django 1.7.1:

如果你想在没有参数的查询中包含文字百分号，你不需要再加上它们。

Changed in Django 1.8:

添加了将参数传递到 `sql` 和 `reverse_sql` 查询的功能。

已添加 `hints` 参数。

`RunSQL.``noop`

New in Django 1.8.

当希望操作不在给定方向执行任何操作时，将 `RunSQL.noop` 属性传递到 `sql` 或 `reverse_sql` 。这在使操作可逆时尤其有用。

RunPython

class `RunPython` (*code*, *reverse_code=None*, *atomic=True*, *hints=None*)

在历史上下文中运行自定义Python代码。 `code` （和 `reverse_code` 如果提供）应该是接受两个参数的可调用对象；第一个是包含与项目历史中的操作位置匹配的历史模型的 `django.apps.registry.Apps` 的实例，第二个是 `SchemaEditor` 的实例。

可选的 `hints` 参数将作为 `**hints` 传递到数据库路由器的 `allow_migrate()` 方法，以帮助他们做出路由决策。有关数据库提示的更多详细信息，请参阅*Hints*。

New in Django 1.8:

已添加 `hints` 参数。

建议您将代码写为迁移文件中 `Migration` 类上方的单独函数，并将其传递到 `RunPython` 。这里有一个使用 `RunPython` 在 `Country` 模型上创建一些初始对象的示例：

```
# -*- coding: utf-8 -*-
from django.db import models, migrations

def forwards_func(apps, schema_editor):
    # We get the model from the versioned app registry;
    # if we directly import it, it'll be the wrong version
    Country = apps.get_model("myapp", "Country")
    db_alias = schema_editor.connection.alias
    Country.objects.using(db_alias).bulk_create([
        Country(name="USA", code="us"),
        Country(name="France", code="fr"),
    ])

class Migration(migrations.Migration):

    dependencies = []

    operations = [
        migrations.RunPython(
            forwards_func,
        ),
    ]
```

这通常是您将用于创建*data migrations*，运行自定义数据更新和更改以及您需要访问ORM和/或Python代码的任何其他操作。

如果你从South升级，这基本上是南模式作为一个操作 - 一个或两个方法向前和向后，可用的ORM和模式操作。大多数情况下，您应该能够翻译 `orm.Model` 或 `orm [“appname”， “Model”]` 引用此处的 `apps.get_model（“appname”， “Model”）` 的数据迁移的代码不变。但是， `apps` 只会引用当前应用程序中的模型，除非将其他应用程序中的迁移添加到迁移的依赖关系中。

与 `RunSQL` 很相似，请确保如果您在此处更改模式，则可以在Django模型系统范围之外（例如，触发器）或使用 `SeparateDatabaseAndState` 添加将反映您对模型状态的更改的操作，否则版本化的ORM和自动检测器将停止正常工作。

默认情况下， `RunPython` 将在事务中运行其内容，即使在不支持DDL事务（例如，MySQL和Oracle）的数据库上。这应该是安全的，但如果您尝试使用这些后端提供的 `schema_editor` ，可能会导致崩溃；在这种情况下，请设置 `atomic=False` 。

警告

`RunPython` 不会神奇地改变模型的连接；您调用的任何模型方法将转到默认数据库，除非您为它们提供当前数据库别名（可从 `schema_editor.connection.alias` 获得，其中 `schema_editor` 是您的第二个参数功能）。

static `RunPython.``noop` ()

New in Django 1.8.

当希望操作在给定方向不执行任何操作时，将 `RunPython.noop` 方法传递到 `code` 或 `reverse_code` 。这在使操作可逆时尤其有用。

SeparateDatabaseAndState

class `SeparateDatabaseAndState` (*database_operations=None*, *state_operations=None*)

高度专业化的操作，允许您混合和匹配数据库（模式更改）和操作的状态（自动检测器供电）方面。

它接受两个操作列表，并且当被要求应用状态时将使用状态列表，并且当被要求对数据库应用更改时将使用数据库列表。不要使用此操作，除非你非常确定你知道你在做什么。

写你自己

操作有一个相对简单的API，它们的设计使您可以轻松地编写自己的内容来补充内置的Django。 `Operation` 的基本结构如下所示：

```
from django.db.migrations.operations.base import Operation

class MyCustomOperation(Operation):

    # If this is False, it means that this operation will be ign
ored by
    # sqlmigrate; if true, it will be run and the SQL collected
for its output.
    reduces_to_sql = False

    # If this is False, Django will refuse to reverse past this
operation.
    reversible = False

    def __init__(self, arg1, arg2):
        # Operations are usually instantiated with arguments in
migration
        # files. Store the values of them on self for later use.
        pass

    def state_forwards(self, app_label, state):
        # The Operation should take the 'state' parameter (an in
stance of
        # django.db.migrations.state.ProjectState) and mutate it
 to match
        # any schema changes that have occurred.
        pass

    def database_forwards(self, app_label, schema_editor, from_s
tate, to_state):
        # The Operation should use schema_editor to apply any ch
anges it
        # wants to make to the database.
        pass

    def database_backwards(self, app_label, schema_editor, from_
state, to_state):
        # If reversible is True, this is called when the operati
on is reversed.
        pass

    def describe(self):
        # This is used to describe what the operation does in co
nsole output.
        return "Custom Operation"
```

您可以使用此模板并使用它，但我们建议您查看 `django.db.migrations.operations` 中的内置Django操作 - 它们很容易阅读，并涵盖了很多示例使用像 `ProjectState` 的迁移框架的半内部方面以及用于获取历史模型的模式。

有些事情要注意：

- 您不需要了解太多关于 `ProjectState` 的信息，只需编写简单的迁移；只需知道它有一个 `apps` 属性，可以访问应用程序注册表（然后您可以调用 `get_model` ）。
- `database_forwards` 和 `database_backwards` 都获得两个状态传递给他们；这些只是代表 `state_forwards` 方法应用的差异，但是为了方便和速度的原因给你。
- `to_state` 中的data*backwards*方法是较旧的_状态；即，一旦迁移已经完成反转将是当前状态的那个。
- 您可能会看到内置操作的 `references_model` 实现；这是自动检测代码的一部分，并且与自定义操作无关。

作为一个简单的例子，让我们做一个加载PostgreSQL扩展（它包含一些PostgreSQL的更令人兴奋的功能）的操作。它很简单；没有模型状态更改，它所做的是运行一个命令：

```
from django.db.migrations.operations.base import Operation

class LoadExtension(Operation):

    reversible = True

    def __init__(self, name):
        self.name = name

    def state_forwards(self, app_label, state):
        pass

    def database_forwards(self, app_label, schema_editor, from_s
tate, to_state):
        schema_editor.execute("CREATE EXTENSION IF NOT EXISTS %s
" % self.name)

    def database_backwards(self, app_label, schema_editor, from_
state, to_state):
        schema_editor.execute("DROP EXTENSION %s" % self.name)

    def describe(self):
        return "Creates extension %s" % self.name
```

模式编辑器

`class BaseDatabaseSchemaEditor[source]`

Django的迁移系统分为两个部分；计算和储存应该执行什么操作的逻辑(`django.db.migrations`)，以及用于把“创建模型”或者“删除字段”变成SQL语句的数据库抽象层 -- 后者是模式编辑器的功能。

你可能并不想像一个普通的开发者使用Django那样，直接和模型编辑器进行交互，但是如果你编写自己的迁移系统，或者有更进一步的需求，这样会比编写SQL语句更方便。

每个Django的数据库后端都提供了它们自己的模式编辑器，并且总是可以通过 `connection.schema_editor()` 上下文管理器来访问。

```
with connection.schema_editor() as schema_editor:
    schema_editor.delete_model(MyModel)
```

它必须通过上下文管理器来使用，因为这样可以管理一些类似于事务和延迟SQL（比如创建 `ForeignKey` 约束）的东西。

它会暴露所有可能的操作作为方法，这些方法应该按照执行修改的顺序调用。可能一些操作或者类型并不可用于所有数据库 -- 例如，MyISAM引擎不支持外键约束。

如果你在为Django编写一个三方的数据库后端，你需要提供 `SchemaEditor` 实现来使用1.7的迁移功能 -- 然而，只要你的数据库在SQL的使用和关系设计上遵循标准，你就应该能够派生Django内建的 `SchemaEditor` 之一，然后简单调整一下语法。同时也要注意，有一些新的数据库特性是迁移所需要的： `can_rollback_ddl` 和 `supports_combined_alters` 都很重要。

方法

execute

`BaseDatabaseSchemaEditor.execute(sql, params=[])[source]`

执行传入的 SQL语句，如果提供了参数则会带上它们。这是对普通数据库游标的一个简单封装，如果用户希望的话，它可以从 `.sql` 文件中获取SQL。

create_model

`BaseDatabaseSchemaEditor.create_model(model)[source]`

为提供的模型在数据库中创建新的表，带有所需的任何唯一性约束或者索引。

delete_model

```
BaseDatabaseSchemaEditor.delete_model(model)[source]
```

删除数据库中的模型的表，以及它带有的任何唯一性约束或者索引。

alter_unique_together

```
BaseDatabaseSchemaEditor.alter_unique_together(model, old_unique_tog
```

修改模型的 `unique_together` 值；这会向模型表中添加或者删除唯一性约束，使它们匹配新的值。

alter_index_together

```
BaseDatabaseSchemaEditor.alter_index_together(model, old_index_toge
```

修改模型的 `index_together` 值；这会向模型表中添加或者删除索引，使它们匹配新的值。

alter_db_table

```
BaseDatabaseSchemaEditor.alter_db_table(model, old_db_table, new_db_
```

重命名模型的表，从 `old_db_table` 变成 `new_db_table` 。

alter_db_tablespace

```
BaseDatabaseSchemaEditor.alter_db_tablespace(model, old_db_tablespac
```

把模型的表从一个表空间移动到另一个中。

add_field

```
BaseDatabaseSchemaEditor.add_field(model, field)[source]
```

向模型的表中添加一列（或者有时几列），表示新增的字段。如果该字段带有 `db_index=True` 或者 `unique=True` ，同时会添加索引或者唯一性约束。

如果字段为 `ManyToManyField并` 且缺少 `through` 值，会创建一个表来表示关系，而不是创建一列。如果提供了 `through` 值，就什么也不做。

如果字段为 `ForeignKey` ，同时会向列上添加一个外键约束。

remove_field

```
BaseDatabaseSchemaEditor.remove_field(model, field)[source]
```

从模型的表中移除代表字段的列，以及列上的任何唯一性约束，外键约束，或者索引。

如果字段是 `ManyToManyField` 并且缺少 `through` 值，会移除创建用来跟踪关系的表。如果提供了 `through` 值，就什么也不做。

alter_field

```
BaseDatabaseSchemaEditor.alter_field(model, old_field, new_field, st
```

这会将模型的字段从旧的字段转换为新的。这包括列名称的修改（ `db_column` 属性）、字段类型的修改（如果修改了字段类）、字段 `NULL` 状态的修改、添加或者删除字段层面的唯一性约束和索引、修改主键、以及修改 `ForeignKey` 约束的目标。

最普遍的一个不能实现的转换，是把 `ManyToManyField` 变成一个普通的字段，反之亦然；Django不能在不丢失数据的情况下执行这个转换，所以会拒绝这样做。作为替代，应该单独调用 `remove_field()` 和 `add_field()` 。

如果数据库满足 `supports_combined_alters` ，Django会尽可能在单次数据库调用中执行所有这些操作。否则对于每个变更，都会执行一个单独的 `ALTER` 语句，但是如果不需要做任何改变，则不执行 `ALTER` （就像South经常做的那样）。

属性

除非另有规定，所有属性都应该是只读的。

connection

```
SchemaEditor.connection
```

一个到数据库的连接对象。 `alias` 是 `connection` 的一个实用的属性，它用于决定要访问的数据库的名字。

当你在多种数据库之间执行迁移的时候，这是非常有用的。

> 译者：Django 文档协作翻译小组，原文：SchemaEditor。
>
>
>
>

编写数据库迁移

这一节介绍你可能遇到的在不同情况下如何分析和编写数据库迁移. 有关迁移的入门资料，请查看 the topic guide.

数据迁移和多数据库

在使用多个数据库时，需要解决是否针对某个特定数据库运行迁移。例如，你可能只 想在某个特定数据库上运行迁移。

为此你可以在RunPython中通过查看schema_editor.connection.alias 属性来检查数据库连接别名：

```
from django.db import migrations

def forwards(apps, schema_editor):
    if not schema_editor.connection.alias == 'default':
        return
    # Your migration code goes here

class Migration(migrations.Migration):

    dependencies = [
        # Dependencies to other migrations
    ]

    operations = [
        migrations.RunPython(forwards),
    ]
```

```
Django 1.8 中新增。
```

你也可以提供一个提示作为 **hints参数传递到数据库路由的allow_migrate() 方法：

```
myapp/dbrouters.py
class MyRouter(object):

    def allow_migrate(self, db, app_label, model_name=None, **hi
nts):
        if 'target_db' in hints:
            return db == hints['target_db']
        return True
```

然后，要在你的迁移中利用，执行以下操作：

```
from django.db import migrations

def forwards(apps, schema_editor):
    # Your migration code goes here

class Migration(migrations.Migration):

    dependencies = [
        # Dependencies to other migrations
    ]

    operations = [
        migrations.RunPython(forwards, hints={'target_db': 'defa
ult'}),
    ]
```

如果你的RunPython或者RunSQL操作只对一个模型有影响，最佳实践是将model_name作为提示传递，使其尽可能对路由可见。这对可复用的和第三方应用极其重要。

添加唯一字段的迁移

如果你应用了一个“朴素”的迁移，向表中一个已存在的行中添加了一个唯一的非空字段，会产生错误，因为位于已存在行中的值只会生成一次。所以需要移除唯一性的约束。

所以，应该执行下面的步骤。在这个例子中，我们会以默认值添加一个非空的UUIDField字段。你可以根据你的需要修改各个字段。

- 把default=...和unique=True参数添加到你模型的字段中。在这个例子中，我们默认使用uuid.uuid4。
- 运行 makemigrations 命令。
- 编辑创建的迁移文件。

生成的迁移类看上去像这样：

```
class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0003_auto_20150129_1705'),
    ]

    operations = [
        migrations.AddField(
            model_name='mymodel',
            name='uuid',
            field=models.UUIDField(max_length=32, unique=True, d
efault=uuid.uuid4),
        ),
    ]
```

你需要做三处更改：

- 从已生成的迁移类中复制，添加第二个AddField操作，并改为AlterField。
- 在第一个AddField操作中，把unique=True改为 null=True，这会创建一个中间的null字段。
- 在两个操作之间，添加一个RunPython或RunSQL操作为每个已存在的行生成一个唯一值（例如UUID）。

最终的迁移类应该看起来是这样：

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import migrations, models
import uuid

def gen_uuid(apps, schema_editor):
    MyModel = apps.get_model('myapp', 'MyModel')
    for row in MyModel.objects.all():
        row.uuid = uuid.uuid4()
        row.save()

class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0003_auto_20150129_1705'),
    ]

    operations = [
        migrations.AddField(
            model_name='mymodel',
            name='uuid',
            field=models.UUIDField(default=uuid.uuid4, null=True
),
        ),
        # omit reverse_code=... if you don't want the migration
to be reversible.
        migrations.RunPython(gen_uuid, reverse_code=migrations.R
unPython.noop),
        migrations.AlterField(
            model_name='mymodel',
            name='uuid',
            field=models.UUIDField(default=uuid.uuid4, unique=Tr
ue),
        ),
    ]
```

现在你可以像平常一样使用migrate命令应用迁移。

注意如果你在这个迁移运行时让对象被创建，就会产生竞争条件(race condition)。在AddField之后， RunPython之前创建的对象会覆写他们原始的uuid。

高级

管理器

class Manager

管理器是一个接口，数据库查询操作通过它提供给django的模型。django应用的每个模型至少拥有一个 管理器。

管理器类的工作方式在 执行查询文档中阐述，而这篇文档涉及了自定义管理器行为的模型选项。

管理器的名字

通常，django为每个模型类添加一个名为objects的管理器。然而，如果你想将objects用于字段名称，或者你想使用其它名称而不是objects访问管理器，你可以在每个模型类中重命名它。在模型中定义一个值为models.Manager()的属性，来重命名管理器。例如：

```
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

使用例子中的模型， Person.objects会抛出AttributeError异常，而Person.people.all()会返回一个包含所有Person对象的列表。

自定义管理器

在一个特定的模型中，你可以通过继承管理器类来构建一个自定义的管理器，以及实例化你的自定义管理器。

你有两个原因可能会自己定义管理器：向器类中添加额外的方法，或者修改管理器最初返回的查询集。

添加额外的管理器方法

为你的模型添加表级(table-level)功能时，采用添加额外的管理器方法是更好的处理方式。如果要添加行级功能——就是说该功能只对某个模型的实例对象起作用。在这种情况下，使用 模型方法 比使用自定义的管理器方法要更好。）

自定义的管理器 方法可以返回你想要的任何数据，而不只是查询集。

例如，下面这个自定义的 管理器提供了一个 with_counts() 方法，它返回所有 OpinionPoll 对象的列表，而且列表中的每个对象都多了一个名为 num_responses 的属性，这个属性保存一个聚合查询(COUNT*)的结果：

```
from django.db import models

class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY p.id, p.question, p.poll_date
            ORDER BY p.poll_date DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date
=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

在这个例子中，你已经可以使用 OpinionPoll.objects.with_counts() 得到所有含有 num_responses属性的 OpinionPoll对象。

这个例子要注意的一点是： 管理器方法可以访问 self.model来得到它所用到的模型类。

修改管理器初始的查询集

管理器自带的 查询集返回系统中所有的对象。例如，使用下面这个模型：

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

... Book.objects.all() 语句将返回数据库中所有的 Book 对象。

你可以通过重写 Manager.get_queryset() 的方法来覆盖 管理器自带的 查询集。get_queryset() 会根据你所需要的属性返回 查询集。

例如，下面的模型有两个 管理器，一个返回所有的对象，另一个则只返回作者是 Roald Dahl 的对象：

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return super(DahlBookManager, self).get_queryset().filte
r(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager
.
```

在这个简单的例子中，Book.objects.all()将返回数据库中所有的图书。而 Book.dahl_objects.all() 只返回作者是 Roald Dahl 的图书。

由于 get_queryset() 返回的是一个 查询集 对象，所以你仍可以对它使用 filter(), exclude()和其他 查询集的方法。所以下面这些例子都是可用的：

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

这个例子还展示了另外一个很有意思的技巧：在同一个模型中使用多个管理器。你可以随你所意在一个模型里面添加多个 Manager() 实例。下面就用很简单的方法，给模型添加通用过滤器：

例如：

```
class AuthorManager(models.Manager):
    def get_queryset(self):
        return super(AuthorManager, self).get_queryset().filter(
role='A')

class EditorManager(models.Manager):
    def get_queryset(self):
        return super(EditorManager, self).get_queryset().filter(
role='E')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices=(('A', _('Auth
or')), ('E', _('Editor'))))
    people = models.Manager()
    authors = AuthorManager()
    editors = EditorManager()
```

在这个例子中，你使用 Person.authors.all(), Person.editors.all(),以及 Person.people.all(), 都会得到和名称相符的结果。

默认管理器

如果你使用了自定义 管理器对象，要注意 Django 中的第一个 管理器 (按照模型中出现的顺序而定) 拥有特殊的地位。Django 会将模型中定义的管理器解释为默认的管理器，并且 Django 中的一部分应用(包括数据备份)会使用默认的管理器，除了前面那个模型。因此，要决定默认的管理器时，要小心谨慎，仔细考量，这样才能避免重写 get_queryset() 导致无法正确地获得数据。

使用管理器访问关联对象

默认情况下，在访问相关对象时（例如choice.poll），Django 并不使用相关对象的默认管理器，而是使用一个"朴素"管理器类的实例来访问。这是因为 Django 要能从关联对象中获得数据，但这些数据有可能被默认管理器过滤掉，或是无法进行访问。

如果普通的朴素管理器类(django.db.models.Manager)并不适用于你的应用，那么你可以通过在管理器类中设置 use_for_related_fields ，强制 Django 在你的模型中使用默认的管理器。这部分内容在 下面有 详细介绍。

调用自定义的查询集

虽然大多数标准查询集的方法可以从管理器中直接访问到，但是这是一个例子，访问了定义在自定义 查询集上的额外方法，如果你也在管理器上面实现了它们：

```
class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role='A')

    def editors(self):
        return self.filter(role='E')

class PersonManager(models.Manager):
    def get_queryset(self):
        return PersonQuerySet(self.model, using=self._db)

    def authors(self):
        return self.get_queryset().authors()

    def editors(self):
        return self.get_queryset().editors()

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices=(('A', _('Auth
or')), ('E', _('Editor'))))
    people = PersonManager()
```

这个例子展示了如何直接从管理器 Person.people调用authors() 和 editors()。

创建管理器

django 1.7 中新增

对于上面的例子，同一个方法需要在查询集 和 管理器上创建两份副本，作为替代，QuerySet.as_manager()可以创建一个管理器的实例，它拥有自定义查询集的方法：

```
class Person(models.Model):
    ...
    people = PersonQuerySet.as_manager()
```

通过QuerySet.as_manager()创建的管理器 实例，实际上等价于上面例子中的PersonManager。

并不是每个查询集的方法都在管理器层面上有意义。比如 QuerySet.delete()，我们有意防止它复制到管理器 中。

方法按照以下规则进行复制：

- 公共方法默认被复制。
- 私有方法（前面带一个下划线）默认不被复制。

- 带queryset_only 属性，并且值为False的方法总是被复制。
- 带 queryset_only 属性，并且值为True 的方法不会被复制。

例如：

```
class CustomQuerySet(models.QuerySet):
    # Available on both Manager and QuerySet.
    def public_method(self):
        return

    # Available only on QuerySet.
    def _private_method(self):
        return

    # Available only on QuerySet.
    def opted_out_public_method(self):
        return
    opted_out_public_method.queryset_only = True

    # Available on both Manager and QuerySet.
    def _opted_in_private_method(self):
        return
    _opted_in_private_method.queryset_only = False
```

from_queryset

```
classmethod from_queryset(queryset_class)
```

在进一步的使用中，你可能想创建一个自定义管理器和一个自定义查询集。你可以调用Manager.from_queryset()，它会返回管理器的一个子类，带有自定义查询集所有方法的副本：

```
class BaseManager(models.Manager):
    def manager_only_method(self):
        return

class CustomQuerySet(models.QuerySet):
    def manager_and_queryset_method(self):
        return

class MyModel(models.Model):
    objects = BaseManager.from_queryset(CustomQueryset)()
```

你也可以在一个变量中储存生成的类：

```
CustomManager = BaseManager.from_queryset(CustomQueryset)

class MyModel(models.Model):
    objects = CustomManager()
```

自定义管理器和模型继承

类继承和模型管理器两者之间配合得并不是很好。 管理器一般只对其定义所在的类起作用，在子类中对其继承绝对不是一个好主意。 而且，因为第一个 管理器会被 Djange 声明为默认的管理器，所以对默认的管理器 进行控制是非常必要的。下面就是 Django 如何处理自定义管理器和模型继承(model inheritance)的：

- 定义在非抽象基类中的管理器是 不会 被子类继承的。如果你想从一个非抽象基类中重用管理器，只能在子类中重定义管理器。 这是因为这种管理器与定义它的模型 绑定得非常紧密，所以继承它们经常会导致异常的结果（特别是默认管理器运行的时候）。 因此，它们不应继承给子类。
- 定义在抽象基类中的管理器总是被子类继续的，是按 Python 的命名解析顺序解析的（首先是子类中的命名覆盖所有的，然后是第一个父类的，以此类推）。 抽象类用来提取子类中的公共信息和行为，定义公共管理器也是公共信息的一部分。
- 如果类当中显示定义了默认管理器，Django 就会以此做为默认管理器；否则就会从第一个抽象基类中继承默认管理器； 如果没有显式声明默认管理器，那么 Django 就会自动添加默认管理器。

如果你想在一组模型上安装一系列自定义管理器，上面提到的这些规则就已经为你的实现提供了必要的灵活性。你可以继承一个抽象基类，但仍要自定义默认的管理器。例如，假设你的基类是这样的：

```
class AbstractBase(models.Model):
    # ...
    objects = CustomManager()

    class Meta:
        abstract = True
```

如果你在基类中没有定义管理器，直接使用上面的代码，默认管理器就是从基类中继承的 objects：

```
class ChildA(AbstractBase):
    # ...
    # This class has CustomManager as the default manager.
    pass
```

如果你想从 AbstractBase继承，却又想提供另一个默认管理器，那么你可以在子类中定义默认管理器：

```
class ChildB(AbstractBase):
    # ...
    # An explicit default manager.
    default_manager = OtherManager()
```

在这个例子中， default_manager就是默认的 管理器。从基类中继承的 objects 管理器仍是可用的。只不过它不再是默认管理器罢了。

最后再举个例子，假设你想在子类中再添加一个额外的管理器，但是很想使用从AbstractBase继承的管理器做为默认管理器。那么，你不在直接在子类中添加新的管理器，否则就会覆盖掉默认管理器，而且你必须对派生自这个基类的所有子类都显示指定管理器。 解决办法就是在另一个基类中添加新的管理器，然后继承时将其放在默认管理器所在的基类 之后。例如：

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()

    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    # ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
    pass
```

注意在抽象模型上面定义一个自定义管理器的时候，不能调用任何使用这个抽象模型的方法。就像：

```
ClassA.objects.do_something()
```

是可以的，但是：

```
AbstractBase.objects.do_something()
```

会抛出一个异常。这是因为，管理器被设计用来封装对象集合管理的逻辑。由于抽象的对象中并没有一个集合，管理它们是毫无意义的。如果你写了应用在抽象模型上的功能，你应该把功能放到抽象模型的静态方法，或者类的方法中。

实现上的注意事项

无论你向自定义管理器中添加了什么功能，都必须可以得到 管理器实例的一个浅表副本：例如，下面的代码必须正常运行：

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django 在一些查询中会创建管理器的浅表副本；如果你的管理器不能被复制，查询就会失败。

这对于大多数自定义管理器不是什么大问题。如果你只是添加一些简单的方法到你的管理器中，不太可能会把你的管理器实例变为不可复制的。但是，如果你覆盖了 __getattr__，或者其它管理器中控制对象状态的私有方法，你应该确保不会影响到管理器的复制。

控制自动管理器的类型

这篇文档已经提到了Django创建管理器类的一些位置：默认管理器和用于访问关联对象的“朴素” 管理器。在 Django 的实现中也有很多地方用到了临时的朴素管理器。 正常情况下，django.db.models.Manager 类的实例会自动创建管理器。

在整个这一节中，我们将那种由 Django 为你创建的管理器称之为 "自动管理器"，既有因为没有管理器而被 Django 自动添加的默认管理器， 也包括在访问关联模型时使用的临时管理器。

有时，默认管理器也并非是自动管理器。 一个例子就是 Django 自带的 django.contrib.gis 应用，所有 gis模型都必须使用一个特殊的管理器类(GeoManager)，因为它们需要运行特殊的查询集(GeoQuerySet)与数据库进行交互。这表明无论自动管理器是否被创建，那些要使用特殊的管理器的模型仍要使用这个特殊的管理器类。

Django 为自定义管理器的开发者提供了一种方式：无论开发的管理器类是不是默认的管理器，它都应该可以用做自动管理器。 可以通过在管理器类中设置 use_for_related_fields 属性来做到这点：

```
class MyManager(models.Manager):
    use_for_related_fields = True
    # ...
```

如果在模型中的默认 管理器(在这些情况中仅考虑默认管理器)中设置了这个属性，那么无论它是否需要被自动创建，Django 都会自动使用它。否则 Django 就会使用 django.db.models.Manager.

> 历史回顾
>
> 从它使用目的来看，这个属性的名称(use_for_related_fields)好象有点古怪。原本，这个属性仅仅是用来控制访问关联字段的管理器的类型，这就是它名字的由来。 后来它的作用更加拓宽了，但是名称一直未变。 因为要保证现在的代码在 Django 以后的版本中仍可以正常工作(continue to work)，这就是它名称不变的原因。

在自动管理器实例中编写正确的管理器

在上面的django.contrib.gis 已经提到了， use_for_related_fields这个特性是在需要返回一个自定义查询集子类的管理器中使用的。要在你的管理器中提供这个功能，要注意以下几点。

不要在这种类型的管理器子类中过滤掉任何结果

一个原因是自动管理器是用来访问关联模型 的对象。 在这种情况下，Django 必须要能看到相关模型的所有对象，所以才能根据关联关系得到任何数据 。

如果你重写了 get_queryset() 方法并且过滤掉了一些行数据，Django 将返回不正确的结果。不要这么做！ 在 get_queryset()方法中过滤掉数据，会使得它所在的管理器不适于用做自动管理器。

设置 use_for_related_fields

use_for_related_fields属性必须在管理器类中设置，而不是在类的 实例中设置。上面已经有例子展示如何正确地设置，下面这个例子就是一个错误的示范：

```
# BAD: Incorrect code
class MyManager(models.Manager):
    # ...
    pass

# Sets the attribute on an instance of MyManager. Django will
# ignore this setting.
mgr = MyManager()
mgr.use_for_related_fields = True

class MyModel(models.Model):
    # ...
    objects = mgr

# End of incorrect code.
```

你也不应该在模型中使用这个属性之后，在类上改变它。这是因为在模型类被创建时，这个属性值马上就会被处理，而且随后不会再读取这个属性值。 这节的第一个例子就是在第一次定义的时候在管理器上设置use_for_related_fields属性，所有的代码就工作得很好。

进行原始的**sql**查询

在模型查询*API*不够用的情况下，你可以使用原始的sql语句。django提供两种方法使用原始sql进行查询：一种是使用**Manager.raw()**方法，进行原始查询并返回模型实例；另一种是完全避开模型层，直接执行自定义的*sql*语句。

> 警告
>
> 编写原始的sql语句时，应该格外小心。每次使用的时候，都要确保转义了参数中的任何控制字符，以防受到sql注入攻击。更多信息请参阅防止*sql*注入。

进行原始查询

raw()方法用于原始的sql查询，并返回模型的实例：

```
Manager.raw(raw_query, params=None, translations=None)
```

这个方法执行原始的sql查询之后，返回**django.db.models.query.RawQuerySet**的实例。**RawQuerySet**实例可以像一般的**QuerySet**那样，通过迭代来提供对象的实例。

这里最好通过例子展示一下，假设存在以下模型：

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

你可以像这样执行自定义的sql语句：

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

当然，这个例子不是特别有趣，和直接使用**Person.objects.all()**的结果一模一样。但是，**raw()**拥有其它更强大的使用方法。

> 模型表的名称
>
> 在上面的例子中，**Person**表的名称是从哪里得到的？
>
> 通常，Django通过将模型的名称和模型的“应用标签”（你在**manage.py startapp**中使用的名称）进行关联，用一条下划线连接他们，来组合表的名称。在这里我们假定**Person**模型存在于一个叫做**myapp**的应用中，所以表就应该叫做**myapp_person**。
>
> 更多细节请查看**db_table**选项的文档，它也可以让你自定义表的名称。
>
> 警告
>
> 传递给**raw()**方法的sql语句并没有任何检查。django默认它会返回一个数据集，但这不是强制性的。如果查询的结果不是数据集，则会产生一个错误。
>
> 警告
>
> 如果你在mysql上执行查询，注意在类型不一致的时候，mysql的静默类型强制可能导致意想不到的结果发生。如果你在一个字符串类型的列上查询一个整数类型的值，mysql会在比较前强制把每个值的类型转成整数。例如，如果你的表中包含值**'abc'**和**'def'**，你查询**'where mycolumn=0'**，那么两行都会匹配。要防止这种情况，在查询中使用值之前，要做好正确的类型转换。
>
> 警告
>
> 虽然**RawQuerySet**可以像普通的**QuerySet**一样迭代，**RawQuerySet**并没有实现可以在**QuerySet**上使用的所有方法。例如，**__bool__()**和**__len__()**在**RawQuerySet**中没有被定义，所以所有**RawQuerySet**转化为布尔值的结果都是**True**。**RawQuerySet**中没有实现他们的原因是，在没有内部缓存的情况下会导致性能下降，而且增加内部缓存不向后兼容。

将查询字段映射到模型字段

raw()方法自动将查询字段映射到模型字段。

字段的顺序并不重要。换句话说，下面两种查询的作用相同：

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_
date FROM myapp_person')
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name
, id FROM myapp_person')
...
```

Django会根据名字进行匹配。这意味着你可以使用sql的**as**子句来映射二者。所以如果在其他的表中有一些**Person**数据，你可以很容易地把它们映射成**Person**实例。

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                              last AS last_name,
...                              bd AS birth_date,
...                              pk AS id,
...                       FROM some_other_table''')
```

只要名字能对应上，模型的实例就会被正确创建。 又或者，你可以在**raw()**方法中使用翻译参数。翻译参数是一个字典，将表中的字段名称映射为模型中的字段名称、例如，上面的查询可以写成这样：

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd'
: 'birth_date', 'pk': 'id'}
>>> Person.objects.raw('SELECT * FROM some_other_table', transla
tions=name_map)
```

索引访问

raw()方法支持索引访问，所以如果只需要第一条记录，可以这样写：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_perso
n')[0]
```

然而，索引和切片并不在数据库层面上进行操作。如果数据库中有很多的**Person**对象，更加高效的方法是在sql层面限制查询中结果的数量：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_perso
n LIMIT 1')[0]
```

延迟加载模型字段

字段也可以被省略：

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myap
p_person')
```

查询返回的Person对象是一个延迟的模型实例（请见 **defer()**）。这意味着被省略的字段，在访问时才被加载。例如:

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myap
p_person'):
...     print(p.first_name, # This will be retrieved by the orig
inal query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

从表面上来看，看起来这个查询获取了**first_name**和**last_name**。然而，这个例子实际上执行了3次查询。只有**first_name**字段在**raw()**查询中获取，**last_name**字符按在执行打印命令时才被获取。

只有一种字段不可以被省略，就是主键。Django 使用主键来识别模型的实例，所以它在每次原始查询中都必须包含。如果你忘记包含主键的话，会抛出一个**InvalidQuery**异常。

增加注解

你也可以在查询中包含模型中没有定义的字段。例如，我们可以使用**PostgreSQL**的**age()**函数来获得一群人的列表，带有数据库计算出的年龄。

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS ag
e FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

向 raw() 方法中传递参数

如果你需要参数化的查询，可以向**raw()** 方法传递**params**参数。

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_na
me = %s', [lname])
```

params是存放参数的列表或字典。你可以在查询语句中使用**%s**占位符，或者对于字典使用**%(key)**占位符（**key**会被替换成字典中键为key的值），无论你的数据库引擎是什么。这样的占位符会被替换成参数表中正确的参数。

> 注意
>
> SQLite后端不支持字典，你必须以列表的形式传递参数。
>
> 警告
>
> 不要在原始查询中使用字符串格式化！
>
> 它类似于这种样子：
>
> ```
> >> query = 'SELECT * FROM myapp_person WHERE last_name = %s'
>  % lname
> >> Person.objects.raw(query)
> ```
>
> 使用参数化查询可以完全防止sql注入，一种普遍的漏洞使攻击者可以向你的数据库中注入任何sql语句。如果你使用字符串格式化，早晚会受到sql输入的攻击。只要你记住默认使用参数化查询，就可以免于攻击。

直接执行自定义sql

有时**Manager.raw()**方法并不十分好用，你不需要将查询结果映射成模型，或者你需要执行**UPDATE**、**INSERT**以及**DELETE**查询。

在这些情况下，你可以直接访问数据库，完全避开模型层。

django.db.connection对象提供了常规数据库连接的方式。为了使用数据库连接，调用**connection.cursor()**方法来获取一个游标对象之后，调用**cursor.execute(sql, [params])**来执行sql语句，调用**cursor.fetchone()**或者**curser.fetchall()**来返回结果行。

例如:

```
from django.db import connection

def my_custom_sql(self):
    cursor = connection.cursor()

    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [sel
f.baz])

    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.b
az])
    row = cursor.fetchone()

    return row
```

注意如果你的查询中包含百分号字符，你需要写成两个百分号字符，以便能正确传递参数：

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id =
%s", [self.id])
```

如果你使用了不止一个数据库，你可以使用**django.db.connections**来获取针对特定数据库的连接（以及游标）对象。**django.db.connections**是一个类似于字典的对象，允许你通过它的别名获取特定的连接

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# Your code here...
```

通常，Python DB API会返回不带字段的结果，这意味着你需要以一个列表结束，而不是一个字典。花费一点性能之后，你可以返回一个字典形式的结果，像这样：

```
def dictfetchall(cursor):
    "Returns all rows from a cursor as a dict"
    desc = cursor.description
    return [
        dict(zip([col[0] for col in desc], row))
        for row in cursor.fetchall()
    ]
```

下面是一个体现二者区别的例子:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
((54360982L, None), (54360880L, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982L}, {'parent_id': None, 'id':
 54360880L}]
```

连接和游标

连接和游标主要实现PEP 249中描述的Python DB API标准，除非它涉及到事务处理。

如果你不熟悉Python DB-API，注意**cursor.execute()**中的sql语句使用占位符**"%s"**，而不是直接在sql中添加参数。如果你使用它，下面的数据库会在必要时自动转义你的参数。

也要注意Django使用**"%s"**占位符，而不是SQLite Python绑定的**"?"**占位符。这是一致性和可用性的缘故。

```
Django 1.7中的改变。
```

PEP 249并没有说明游标是否可以作为上下文管理器使用。在python2.7之前，游标可以用作上下文管理器，由于魔术方法lookups中意想不到的行为(Python ticket #9220)。Django 1.7 显式添加了对允许游标作为上下文管理器使用的支持。

将游标作为上下文管理器使用:

```
with connection.cursor() as c:
    c.execute(...)
```

等价于：

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

数据库事务

Django 为你提供几种方法来控制如何管理数据库事务。

管理数据库事务

Django’s default transaction behavior

Django 的默认行为是运行在自动提交模式下。任何一个查询都立即被提交到数据库中，除非激活一个事务。具体细节看下面.

Django 用事务或者保存点去自动的保证复杂ORM各种查询操作的统一性,尤其是 *delete()* 和 *update()* 查询.

Django’s `测试用例` 也包装了事务性能原因的测试类

把事务绑定到HTTP 请求上

在web上一种简单处理事务的方式是把每个请求用事务包装起来.在每个你想保存这种行为的数据库的配置文件中，设置 `ATOMIC_REQUESTS` 值为 `True`，

它是这样工作的。在调用一个view里面的方法之前，django开始一个事务如果发出的响应没有问题,Django就会提交这个事务。如果在view这里产生一个异常，Django就会回滚这次事务

你可能会在你的视图代码中执行一部分提交并且回滚，通常使用 `atomic()` context管理器.但是最后你的视图，要么是所有改变都提交执行，要么是都不提交。

警告

虽然这种简洁的事物模型看上去很吸引人, 但要注意当流量增长时它会表现出较差的效率。对每个视图开启一个事务是有所耗费的。其对性能的影响依赖于应用程序对数据库的查询语句效率和数据库当前的锁竞争情况。

预请求事务和流式响应

当一个视图返回一个 `StreamingHttpResponse` 时, 其读取的内容是由执行代码来产生的。因为视图调用已经返回，这样代码在事务的外部运行。

一般而言，在产生一个流式响应时，不建议再进行写数据库的操作，因为没有明智的方式在开始发送响应之后来处理错误。

在实际操作时，可以通过如下 `atomic()` 装饰器把这一功能简单地加载到视图函数上。?

表示事务仅仅是在当前视图中有效，诸如模板响应之类的中间件(Middleware)操作是运行在事务之外的。

当 `ATOMIC_REQUESTS` 被启用后，仍然有办法来阻止视图运行一个事务操作。

`non_atomic_requests` (*using=None*)[source]

这个装饰器会否定一个由 `ATOMIC_REQUESTS` 设定的视图:

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    do_stuff()

@transaction.non_atomic_requests(using='other')
def my_other_view(request):
    do_stuff_on_the_other_database()
```

它将仅工作在设定了此装饰器的视图上。

更加明确地控制事务

Django提供了单一的API来控制数据库事务。

`atomic` (*using=None*, *savepoint=True*)[source]

原子性是由数据库的事务操作来界定的。 `atomic` 允许我们在执行代码块时，在数据库层面提供原子性保证。 如果代码块成功完成， 相应的变化会被提交到数据库进行commit；如果执行期间遇到异常，则会将该段代码所涉及的所有更改回滚。.

`atomic` 块可以嵌套。 在下面的例子里，使用with语句，当一个内部块完成后，如果某个异常在外部块被抛出，内部块上的操作仍然可以回滚(前提是外部块也被atomic装饰过)。

`atomic` 被用作装饰器:

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```

atomic被用作 上下文管理器:

```
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

经过 `atomic` 装饰的代码在一个 try/except 块内允许使用常见的完整性错误检测语法:

```
from django.db import IntegrityError, transaction

@transaction.atomic
def viewfunc(request):
    create_parent()

    try:
        with transaction.atomic():
            generate_relationships()
    except IntegrityError:
        handle_exception()

    add_children()
```

在这个例子中，即使 `generate_relationships()` 违反完整性约束导致了数据库错误， 你仍可以进行 `add_children()` 的操作, 并且 `create_parent()` 的变化仍然存在。注意，当? `handle_exception()` 被触发时，在 `generate_relationships()` 上的尝试操作已经被安全回滚，所以若有必要，这个异常的句柄也能够操作数据库。

避免在 `atomic` 里捕获异常!

当一个 `原子块` 执行完退出时，Django会审查是正常提交还是回滚。如果你在 `原子` 块中捕获了异常的句柄, 你可能就向Django隐藏了问题的发生。这可能会导致意想不到的后果。

这主要是考虑到 `DatabaseError` 和其诸如 `IntegrityError` 这样的子类。 若是遇到这样的错误，事务的原子性会被打破，Django会在 `原子` 代码块上执行回滚操作。如果你试图在回滚发生前运行数据库查询，Django会产生一个 `TransactionManagementError` 的异常。当一个ORM-相关的信号句柄操作异常时，你可能也会遇到类似的情形。

正确捕捉数据库异常应该是类似上文所讲 ，基于 `atomic` 代码块来做。若有必要, 可以额外增加一层 `atomic` 代码来用于此目的。这种模式还有另一个优势：它明确了当一个异常发生时，哪些操作将回滚。

如果你是从原始的SQL查询语句中捕获异常，则Django的行为是不明确的，而且是依赖于数据库的。

为了确保原子性达成, `atomic` 会 disables一些 APIs. 在 `atomic` 代码块内试图 commit, roll back,或者更改数据库autocommit的状态都会导致异常。

`atomic` 使用的 `using` 参数必须是数据库的名字. 如果这个参数没提供, Django 默认使用 `"default"` 数据库。

在底层，Django的事务管理代码：

- 当进入到最外层的 `atomic` 代码块时会打开一个事务;
- 当进入到内层 `atomic` 代码块时会创建一个保存点;
- 当退出内部块时会释放或回滚保存点;
- 当退出外部块时提交或回退事物。

你可以通过设置 `savepoint` 参数为 `False` 来使对内层的保存点失效。如果异常发生，若设置了savepoint，Django会在退出第一层代码块时执行回滚，否则会在最外层的代码块上执行回滚。 原子性始终会在外层事物上得到保证。这个选项仅仅用在设置保存点开销很明显时的情况下。它的缺点是打破了上述错误处理的原则。

在autocommit关闭的情况下，你可以使用 `atomic` . 这只使用savepoints功能，即使是对于最外层的块。如果在最外层的块上声明 `savepoint=False` ，这将会产生一个错误。

性能考虑

所有打开的事务会对数据库带来性能成本。要尽量减少这种开销，尽量保持您的交易尽可能短。 在Django的请求/响应周期，如果你使用? `atomic()` 来执行长运行的进程，这尤其重要。

Autocommit

为什么 Django会使用autocommit

在SQL标准中, 每个SQL语句在执行时都会启动一个事务，除非已经存在一个事务了。这样事务必须明确是提交还是回滚。

对应用程序开发者而言，这样非常不方便。为了避免这个问题，大多数数据库提供了一个autocommit模式。当 autocommit被打开并且事物处于活动状态时，每个SQL查询都可以看成是一个事物。也就是说, 不但每个查询是每个事物的开始，而且每个事物会自动提交或回滚，这取决于该查询是否成功执行。

PEP 249, Python数据库API 规范v2.0, 需要将autocommit初试设置为关闭状态。Django覆盖了这个默认规范并且将autocommit设置为 on.

要想避免这样, 你可以关闭事务管理器, 但不建议这样做。

关闭事务管理器

你可以在配置文件里通过设置 `AUTOCOMMIT` 为? `False` 完全关闭Django的事物管理。如果这样做了，Django将不能启用autocommit,也不能执行任何 commits. 你只能遵照数据库层面的规则行为。

这就需要你对每个事物执行明确的commit操作，即使由Django或第三方库创建的。因此，这最好只用于你自定义的事物控制中间件或者是一些比较奇特的场景。

更低级别的API

警告

如果可能的话，尽量优先选择 `atomic()` 来控制事物 ，它遵守数据库的相关特性并且防止了非法操作。

低级别 API仅仅用于你自定义的事物管理场景。

Autocommit

在 `django.db.transaction` 模块里，Django提供了一个简单的API， 用于管理每个数据库的自提交状态。

`get_autocommit` (*using=None*)[source]

`set_autocommit` (*autocommit*, *using=None*)[source]

这些函数使用了一个 `using` 参数，参数的值是数据库的名字。 如果参数没有提供, Django使用 `"default"` 数据库。

Autocommit初始是打开的。如果你把它关掉了,那么你有义务恢复它。

一旦你把autocommit 关掉了, 那么你得到就是数据库的默认行为， Django 不会帮你做任何事。虽然在 **PEP 249**有描述此规范,但数据库适配器的实现并不总与规范是一致的。请仔细检查你当前正在使用的数据库适配器文档。

在把autocommit改回on之前，你必须确保所有的SQL事物处于非活跃状态，通常是使用 `commit()` 或者 `rollback()` 这样的语法操作。

当 `atomic()` 代码块处于活跃状态时，Django会拒绝将autocommit从on的状态调整为off，因为这样会破坏原子性。?

事务

事务是一系列数据库语句的原子集。即使程序在运行时崩溃了,数据库可以确保事物集中的所有变更要么都被提交，要么都被放弃。

Django 并没有提供一个API来开启一个事务。因为开始事务的预期方式是将 `set_autocommit()` 设置为disable状态。

一旦你处于一个事物之中，你可以选择要么apply所有变更? `commit()` 提交它，要么取消所有变化 `rollback()` . 这个函数功能是在 `django.db.transaction` 定义的。

`commit` (*using=None*)[source]

`rollback` (*using=None*)[source]

这个函数通过 `using` 指定数据库的名字作为参数。如果没提供, Django 使用 `"default"` 数据库.

当一个? `atomic()` 程序块在运行状态，Django会拒绝 commit 或rollback操作，因为这些操作是自动的。

Savepoints

保存点是在事物执行过程中的一个标记，它可以让你执行回滚事物的一部分 ，而不是整个事物。 保存点在 SQLite (≥ 3.6.8), PostgreSQL, Oracle和MySQL (当使用nnoDB存储引擎时)是有效的。在其他的数据库后端虽然也提供保存点的函数，但其实它们是空操作，实际不起任何作用。

如果你开启了autocommit，Savepoints并没有太多用处，因为这是Django的默认行为。然而, 一旦你使用 `atomic()` 开启了一个事物, 那么你所建立的一系列数据库操作将被视为一个整体，等待同时提交或回滚。 如果你触发了一个回滚,那么整个事物就要进行回滚。Savepoints提供了更细粒度的回滚，而不是用 `transaction.rollback()` 对整个事物进行回滚.

当嵌套使用 `atomic()` 装饰器时, 它会创建 savepoint以允许部分提交或回滚。 强烈建议你使用 `atomic()` 而不是下面描述的函数功能，当然他们也是公开API的一部分，并且现在也没有废除它们的计划。

每个函数都带一个 `using` 参数，这个参数是你要操作的数据库的名字。 如果没有 `using` 参数，则会使用 `"default"` 数据库。

Savepoints 是由 `django.db.transaction` 里的三个函数来控制的：

`savepoint` (*using=None*)[source]

创建一个新的保存点。这将实现在事物里对“好”的状态做一个标记点。返回值是 savepoint ID (`sid`).

`savepoint_commit` (*sid*, *using=None*)[source]

释放保存点 `sid` . 自创建保存点进行的更改将成为事物的一部分。

`savepoint_rollback` (*sid*, *using=None*)[source]

回滚事物保存点 `sid` .

如果不支持保存点或者数据库未处于autocommit模式，这些函数将什么也不做。

此外，还有一个实用的功能：

`clean_savepoints` (*using=None*)[source]

重置用来生成唯一保存点ID的计数器：

下面的例子演示了如何使用保存点：

```
from django.db import transaction

# open a transaction
@transaction.atomic
def viewfunc(request):

    a.save()
    # transaction now contains a.save()

    sid = transaction.savepoint()

    b.save()
    # transaction now contains a.save() and b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # open transaction still contains a.save() and b.save()
    else:
        transaction.savepoint_rollback(sid)
        # open transaction now contains only a.save()
```

保存点通过实现部分回滚实现对数据库报错的恢复。如果你是在一个 `atomic()` 块中这么干的话，那整个块都会被回滚，因为Django并不知你在下一层做此处理操作。为了避免这样，你可以在下面函数中控制回滚行为。

`get_rollback` (*using=None*)[source]

`set_rollback` (*rollback*, *using=None*)[source]

当退出最内层atomic块时设置回滚标记为 `True` 以实现强制回滚。这对在没有抛出异常的情况下触发一个回滚操作是很有用的。

将标志设为 `False` 阻止这样一个回滚。 在此之前，请确保你已经把事物回滚到了该原子块内一个已知良好的保存点。否则，你打破原子性，并且数据损坏可能会发生。

具体数据库的相关说明

Savepoints in SQLite

?SQLite ≥ 3.6.8之后开始支持savepoints,但由于 `sqlite3` 模块的设计缺陷导致其很难使用。

当自动提交被启用，保存点是没有意义的. 当被禁用时， `sqlite3` 在savepoint之前已经进行了的隐式的提交。(实际上, 在任何诸如 `SELECT` , `INSERT` , `UPDATE` , `DELETE` 和 `REPLACE` 等操作语句之前都会进行提交.) 这个问题有两个后果：

- savepoint的低级别API只能用于内部事物。在一个 `atomic()` 块之内。
- 当autocommit处于关闭状态时，是不可能使用 `atomic()` 的。

MySQL中的事务

如果你正在使用 MySQL, 你的表也许支持事务，也许不支持。这具体依赖于你的MySQL版本和你正在使用的表的engine类型。(这里的表engine类型是指“InnoDB” 或 “MyISAM”等.) MySQL的事物特性不在本文讨论的范围之内，你可以从MySQL的官方站点获取MySQL事物的相关信息.

如果你安装的MySQL 不支持事务, 那么Django会一直工作在自动提交模式：语句一旦被调用就会被执行和提交。 如果你安装的MySQL确定支持事物, Django会遵循本文所介绍的关于事务的处理原则。

处理PostgreSQL的交易中的异常

注意

本节内容只有当你实现你自己的事务管理时才相关。 这个问题不会出现在 Django 默认模式和 `atomic()` 自动控制的情况下。

在一个事物内部, 当调用一个PostgreSQL光标抛出一个异常 (通常是 `IntegrityError`), 后续所有在此同一个事物中的SQL将失败并报以下错误“current transaction is aborted, queries ignored until end of transaction block”. 虽然有时简单用 `save()` 是不太可能导致PostgreSQL异常的, 但仍然有更高级模式用法的可能, 例如在一个有唯一约束的字段保存对象, 保存使用force_insert/force_update 标志,或者调用一些定制化的SQL。

有几种方法可以从这种错误中恢复过来。

事务回滚

第一个选择是回滚整个事物。例如：

```
a.save() # Succeeds, but may be undone by transaction rollback
try:
    b.save() # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save() # Succeeds, but a.save() may have been undone
```

调用 `transaction.rollback()` 回滚整个事物。任何未提交的数据库操作都会丢失。在此例中, 由 `a.save()` 所保存的变更将会丢失,即使这个操作自身没有产生错误。

保存点回滚

你可以使用 *savepoints*来控制一个回滚的扩展。 在执行数据库操作可能失败之前，你可以设置或更新保存点； 这样，如果操作失败，您可以回滚该单违规操作，而不是整个事务。例如：

```
a.save() # Succeeds, and never undone by savepoint rollback
sid = transaction.savepoint()
try:
    b.save() # Could throw exception
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save() # Succeeds, and a.save() is never undone
```

在此例中, 当 `b.save()` 抛出异常的情况下， `a.save()` 所做的更改将不会丢失。

聚合

Django数据库抽象API描述了使用Django查询来增删查改单个对象的方法。然而，你有时候会想要获取从一组对象导出的值或者是聚合一组对象。这份指南描述了通过Django查询来生成和返回聚合值的方法。

整篇指南我们都将引用以下模型。这些模型用来记录多个网上书店的库存。

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

class Publisher(models.Model):
    name = models.CharField(max_length=300)
    num_awards = models.IntegerField()

class Book(models.Model):
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    pubdate = models.DateField()

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
    registered_users = models.PositiveIntegerField()
```

速查表

急着用吗？以下是在上述模型的基础上，进行一般的聚合查询的方法:

```
# Total number of books.
>>> Book.objects.count()
2452

# Total number of books with publisher=BaloneyPress
>>> Book.objects.filter(publisher__name='BaloneyPress').count()
73

# Average price across all books.
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}

# Max price across all books.
>>> from django.db.models import Max
>>> Book.objects.all().aggregate(Max('price'))
{'price__max': Decimal('81.20')}

# Cost per page
>>> Book.objects.all().aggregate(
...    price_per_page=Sum(F('price')/F('pages'), output_field=Fl
oatField()))
{'price_per_page': 0.4470664529184653}

# All the following queries involve traversing the Book<->Publis
her
# many-to-many relationship backward

# Each publisher, each with a count of books as a "num_books" at
tribute.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count('book'))
>>> pubs
[<Publisher BaloneyPress>, <Publisher SalamiPress>, ...]
>>> pubs[0].num_books
73

# The top 5 publishers, in order by number of books.
>>> pubs = Publisher.objects.annotate(num_books=Count('book')).o
rder_by('-num_books')[:5]
>>> pubs[0].num_books
1323
```

在查询集上生成聚合

Django提供了两种生成聚合的方法。第一种方法是从整个查询集生成统计值。比如，你想要计算所有在售书的平均价钱。Django的查询语法提供了一种方式描述所有图书的集合。

```
>>> Book.objects.all()
```

我们需要在QuerySet.对象上计算出总价格。这可以通过在QuerySet后面附加aggregate() 子句来完成。

```
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}
```

all()在这里是多余的，所以可以简化为：

```
>>> Book.objects.aggregate(Avg('price'))
{'price__avg': 34.35}
```

aggregate()子句的参数描述了我们想要计算的聚合值，在这个例子中，是Book 模型中price字段的平均值。查询集参考中列出了聚合函数的列表。

aggregate()是QuerySet 的一个终止子句，意思是说，它返回一个包含一些键值对的字典。键的名称是聚合值的标识符，值是计算出来的聚合值。键的名称是按照字段和聚合函数的名称自动生成出来的。如果你想要为聚合值指定一个名称，可以向聚合子句提供它。

```
>>> Book.objects.aggregate(average_price=Avg('price'))
{'average_price': 34.35}
```

如果你希望生成不止一个聚合，你可以向aggregate()子句中添加另一个参数。所以，如果你也想知道所有图书价格的最大值和最小值，可以这样查询：

```
>>> from django.db.models import Avg, Max, Min
>>> Book.objects.aggregate(Avg('price'), Max('price'), Min('price'))
{'price__avg': 34.35, 'price__max': Decimal('81.20'), 'price__min': Decimal('12.99')}
```

为查询集的每一项生成聚合

生成汇总值的第二种方法，是为QuerySet中每一个对象都生成一个独立的汇总值。比如，如果你在检索一列图书，你可能想知道有多少作者写了每一本书。每本书和作者是多对多的关系。我们想要汇总QuerySet.中每本书里的这种关系。

逐个对象的汇总结果可以由annotate()子句生成。当annotate()子句被指定之后，QuerySet中的每个对象都会被注上特定的值。

这些注解的语法都和aggregate()子句所使用的相同。annotate()的每个参数都描述了将要被计算的聚合。比如，给图书添加作者数量的注解：

```
# Build an annotated queryset
>>> from django.db.models import Count
>>> q = Book.objects.annotate(Count('authors'))
# Interrogate the first object in the queryset
>>> q[0]
<Book: The Definitive Guide to Django>
>>> q[0].authors__count
2
# Interrogate the second object in the queryset
>>> q[1]
<Book: Practical Django Projects>
>>> q[1].authors__count
1
```

和使用 aggregate()一样，注解的名称也根据聚合函式的名称和聚合字段的名称得到的。你可以在指定注解时，为默认名称提供一个别名：

```
>>> q = Book.objects.annotate(num_authors=Count('authors'))
>>> q[0].num_authors
2
>>> q[1].num_authors
1
```

与 aggregate() 不同的是， annotate() 不是一个终止子句。annotate()子句的返回结果是一个查询集 (QuerySet)；这个 QuerySet可以用任何QuerySet方法进行修改，包括 filter(), order_by(), 甚至是再次应用annotate()。

> 有任何疑问的话，请检查 SQL query！
>
> 要想弄清楚你的查询到底发生了什么，可以考虑检查你QuerySet的 query 属性。
>
> 例如，在annotate() 中混入多个聚合将会得出错误的结果，因为多个表上做了交叉连接，导致了多余的行聚合。

连接和聚合

至此，我们已经了解了作用于单种模型实例的聚合操作， 但是有时，你也想对所查询对象的关联对象进行聚合。

在聚合函式中指定聚合字段时，Django 允许你使用同样的 双下划线 表示关联关系，然后 Django 在就会处理要读取的关联表，并得到关联对象的聚合。

例如，要得到每个书店的价格区别，可以使用如下注解：

```
>>> from django.db.models import Max, Min
>>> Store.objects.annotate(min_price=Min('books__price'), max_pr
ice=Max('books__price'))
```

这段代码告诉 Django 获取书店模型，并连接(通过多对多关系)图书模型，然后对每本书的价格进行聚合，得出最小值和最大值。

同样的规则也用于 aggregate() 子句。如果你想知道所有书店中最便宜的书和最贵的书价格分别是多少：

```
>>> Store.objects.aggregate(min_price=Min('books__price'), max_p
rice=Max('books__price'))
```

关系链可以按你的要求一直延伸。 例如，想得到所有作者当中最小的年龄是多少，就可以这样写：

```
>>> Store.objects.aggregate(youngest_age=Min('books__authors__ag
e'))
```

遵循反向关系

和 跨关系查找的方法类似，作用在你所查询的模型的关联模型或者字段上的聚合和注解可以遍历"反转"关系。关联模型的小写名称和双下划线也用在这里。

例如，我们可以查询所有出版商，并注上它们一共出了多少本书（注意我们如何用 'book'指定Publisher -> Book 的外键反转关系）：

```
>>> from django.db.models import Count, Min, Sum, Avg
>>> Publisher.objects.annotate(Count('book'))
```

QuerySet结果中的每一个Publisher都会包含一个额外的属性叫做book__count。

我们也可以按照每个出版商，查询所有图书中最旧的那本：

```
>>> Publisher.objects.aggregate(oldest_pubdate=Min('book__pubdat
e'))
```

（返回的字典会包含一个键叫做 'oldest_pubdate'。如果没有指定这样的别名，它会更长一些，像 'book**pubdate**min'。）

这不仅仅可以应用挂在外键上面。还可以用到多对多关系上。例如，我们可以查询每个作者，注上它写的所有书（以及合著的书）一共有多少页（注意我们如何使用 'book'来指定Author -> Book的多对多的反转关系）：

```
>>> Author.objects.annotate(total_pages=Sum('book__pages'))
```

（每个返回的QuerySet中的Author 都有一个额外的属性叫做total_pages。如果没有指定这样的别名，它会更长一些，像 book**pages**sum。）

或者查询所有图书的平均评分，这些图书由我们存档过的作者所写：

```
>>> Author.objects.aggregate(average_rating=Avg('book__rating'))
```

（返回的字典会包含一个键叫做'average**rating**'。如果没有指定这样的别名，它会更长一些，像'**book**rating__avg'。）

聚合和其他查询集子句

filter() 和 exclude()

聚合也可以在过滤器中使用。 作用于普通模型字段的任何 filter()(或 exclude()) 都会对聚合涉及的对象进行限制。

使用annotate() 子句时，过滤器有限制注解对象的作用。例如，你想得到以 "Django" 为书名开头的图书作者的总数：

```
>>> from django.db.models import Count, Avg
>>> Book.objects.filter(name__startswith="Django").annotate(num_
authors=Count('authors'))
```

使用aggregate()子句时，过滤器有限制聚合对象的作用。例如，你可以算出所有以 "Django" 为书名开头的图书平均价格：

```
>>> Book.objects.filter(name__startswith="Django").aggregate(Avg
('price'))
```

对注解过滤

注解值也可以被过滤。 像使用其他模型字段一样，注解也可以在filter()和exclude() 子句中使用别名。

例如，要得到不止一个作者的图书，可以用：

```
>>> Book.objects.annotate(num_authors=Count('authors')).filter(n
um_authors__gt=1)
```

这个查询首先生成一个注解结果，然后再生成一个作用于注解上的过滤器。

annotate() 的顺序

编写一个包含 annotate() 和 filter() 子句的复杂查询时，要特别注意作用于 QuerySet的子句的顺序。

当一个annotate() 子句作用于某个查询时，要根据查询的状态才能得出注解值，而状态由 annotate() 位置所决定。以这就导致filter() 和 annotate() 不能交换顺序，下面两个查询就是不同的：

```
>>> Publisher.objects.annotate(num_books=Count('book')).filter(book__rating__gt=3.0)
```

另一个查询：

```
>>> Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count('book'))
```

两个查询都返回了至少出版了一本好书(评分大于 3 分)的出版商。 但是第一个查询的注解包含其该出版商发行的所有图书的总数；而第二个查询的注解只包含出版过好书的出版商的所发行的图书总数。 在第一个查询中，注解在过滤器之前，所以过滤器对注解没有影响。 在第二个查询中，过滤器在注解之前，所以，在计算注解值时，过滤器就限制了参与运算的对象的范围。

order_by()

注解可以用来做为排序项。 在你定义 order_by() 子句时，你提供的聚合可以引用定义的任何别名做为查询中 annotate()子句的一部分。

例如，根据一本图书作者数量的多少对查询集 QuerySet进行排序：

```
>>> Book.objects.annotate(num_authors=Count('authors')).order_by('num_authors')
```

values()

通常，注解会添加到每个对象上 —— 一个被注解的QuerySet会为初始QuerySet的每个对象返回一个结果集。但是，如果使用了values()子句，它就会限制结果中列的范围，对注解赋值的方法就会完全不同。不是在原始的 QuerySet返回结果中对

每个对象中添加注解，而是根据定义在values() 子句中的字段组合对先结果进行唯一的分组，再根据每个分组算出注解值， 这个注解值是根据分组中所有的成员计算而得的：

例如，考虑一个关于作者的查询，查询出每个作者所写的书的平均评分：

```
>>> Author.objects.annotate(average_rating=Avg('book__rating'))
```

这段代码返回的是数据库中所有的作者以及他们所著图书的平均评分。

但是如果你使用了values()子句，结果是完全不同的：

```
>>> Author.objects.values('name').annotate(average_rating=Avg('b
ook__rating'))
```

在这个例子中，作者会按名称分组，所以你只能得到某个唯一的作者分组的注解值。 这意味着如果你有两个作者同名，那么他们原本各自的查询结果将被合并到同一个结果中；两个作者的所有评分都将被计算为一个平均分。

annotate() 的顺序

和使用 filter() 子句一样，作用于某个查询的annotate() 和 values() 子句的使用顺序是非常重要的。如果values() 子句在 annotate() 之前，就会根据 values() 子句产生的分组来计算注解。

但是，如果 annotate() 子句在 values()子句之前，就会根据整个查询集生成注解。在这种情况下，values() 子句只能限制输出的字段范围。

举个例子，如果我们互换了上个例子中 values()和 annotate() 子句的顺序：

```
>>> Author.objects.annotate(average_rating=Avg('book__rating')).
values('name', 'average_rating')
```

这段代码将给每个作者添加一个唯一的字段，但只有作者名称和average_rating 注解会返回在输出结果中。

你也应该注意到 average_rating 显式地包含在返回的列表当中。之所以这么做的原因正是因为values() 和 annotate() 子句。

如果 values() 子句在 annotate() 子句之前，注解会被自动添加到结果集中；但是，如果 values() 子句作用于annotate() 子句之后，你需要显式地包含聚合列。

与默认排序或order_by()交互

在查询集中的order_by() 部分(或是在模型中默认定义的排序项) 会在选择输出数据时被用到，即使这些字段没有在values() 调用中被指定。这些额外的字段可以将相似的数据行分在一起，也可以让相同的数据行相分离。在做计数时，就会表现地格外明显：

通过例子中的方法，假设有一个这样的模型：

```
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length=10)
    data = models.IntegerField()

    class Meta:
        ordering = ["name"]
```

关键的部分就是在模型默认排序项中设置的name字段。如果你想知道每个非重复的data值出现的次数，可以这样写：

```
# Warning: not quite correct!
Item.objects.values("data").annotate(Count("id"))
```

...这部分代码想通过使用它们公共的 data 值来分组 Item对象，然后在每个分组中得到 id 值的总数。但是上面那样做是行不通的。这是因为默认排序项中的 name也是一个分组项，所以这个查询会根据非重复的 (data, name) 进行分组，而这并不是你本来想要的结果。所以，你应该这样改写：

```
Item.objects.values("data").annotate(Count("id")).order_by()
```

...这样就清空了查询中的所有排序项。 你也可以在其中使用 data ，这样并不会有副作用，这是因为查询分组中只有这么一个角色了。

这个行为与查询集文档中提到的 distinct() 一样，而且生成规则也一样：一般情况下，你不想在结果中由额外的字段扮演这个角色，那就清空排序项，或是至少保证它仅能访问 values()中的字段。

> 注意
>
> 你可能想知道为什么 Django 不删除与你无关的列。主要原因就是要保证使用distinct()和其他方法的一致性。Django 永远不会 删除你所指定的排序限制(我们不能改动那些方法的行为，因为这会违背 API stability 原则)。

聚合注解

你也可以在注解的结果上生成聚合。 当你定义一个 aggregate() 子句时，你提供的聚合会引用定义的任何别名做为查询中 annotate() 子句的一部分。

例如，如果你想计算每本书平均有几个作者，你先用作者总数注解图书集，然后再聚合作者总数，引入注解字段：

```
>>> from django.db.models import Count, Avg
>>> Book.objects.annotate(num_authors=Count('authors')).aggregat
e(Avg('num_authors'))
{'num_authors__avg': 1.66}
```

编写自定义 model 字段

介绍

*model 参考*文档已经介绍了如何使用 Django 的标准字段类；例如 `CharField` ， `DateField` ，等等。对于很多应用来说，这些类足够用了。 但是在某些情况下，你所用的Django 不具备你需要的某些精巧功能，或是你想使用的字段与 Django 自带字段完全不同。

Django 内置的字段类型并不能覆盖所有可能遇到的数据库的列类型，仅仅是些普通的字段类型，例如 `VARCHAR` 和 `INTEGER` 。对于更多不常用的列类型，比如地理定位数据和诸如PostgreSQL自定义类型的自定义字段，你可以定义你自己的 Django `Field` 子类。

有两种实现方式：你可以编写一个复杂的 Python 对象，让它以某种方式将数据序列化，以适应某个数据库的列类型； 或是你创建一个 `Field` 的子类，从而让你可以使用 model 中的对象。

示例对象

创建自定义字段需要注意很多细节。 为了使这一章内容容易理解，自始至终我们都只使用这一个例子：包装一个 Python 对象来表示手中桥牌的详细信息。不用担心，这个例子并不要求你会玩桥牌。 你只要知道 52 张牌被平均分配给四个玩家，按惯例，他们被称之为北, 东, 南 和 西。我们的类看起来就象这个样子：

```
class Hand(object):
    """A hand of cards (bridge style)"""

    def __init__(self, north, east, south, west):
        # Input parameters are lists of cards ('Ah', '9s', etc)
        self.north = north
        self.east = east
        self.south = south
        self.west = west

    # ... (other possibly useful methods omitted) ...
```

这只是一个普通的 Python 类，并没有对 Django 做特别的设定。 在 model 中我们可以象下面这样使用 Hand (我们假设 model 中的 `hand` 属性是 `Hand` 类的一个实例)：

```
example = MyModel.objects.get(pk=1)
print(example.hand.north)

new_hand = Hand(north, east, south, west)
example.hand = new_hand
example.save()
```

我们就象使用任何 Python 类一样，对 model 中的 `hand` 属性进行赋值和取值。利用这一点让 Django 知道如何处理保存和载入一个对象。

为了在 model 中使用 `Hand` 类，我们不必为这个类做任何的改动。这是非常有用的，它表示着你可以很容易地为已存在的类编写模型支持，而不必改动类的原代码。

注意

你可能只想利用自定义数据库列类型，将数据处理成模型中的标准 Python 类型，比如：字符串，浮点数等等。 这种情况与我们的 `Hand` 例子非常相似，我们随着文档的展开对两者的差异进行比较。

后台原理

数据库存储

可以简单的认为 model 字段提供了一种方法来接受普通的 Python 对象，比如布尔值， `datetime` ，或是象 `Hand` 这样更复杂的对象，然后在操作数据库时，对对象进行格式转换以适应数据库。（还有序列化也是同样处理，但接下来我们会看到，一旦我们掌握了数据库这方面的转换，再对序列化做处理就游刃有余了）

模型中的字段必须以某种方式转化为现有的数据库字段类型。不同的数据库提供了不同的有效的列类型的集合，但是规则仍然是相同的：这些是唯一工作类型。任何你想存储在数据库中必须适应这些类型之一。

通常情况下，你可以写一个Django字段来匹配特定的数据库行的类型，或有一个相当直接的的方式将你的数据转化为一个字符串。

以我们的 `Hand` 为例，我们可以将卡片的数据以预先决定好的顺序，连接转化为一个104个字符的字符串– 也就是说， *north* 卡片排在第一, 然后是 *east*, *south* 和 *west* 。所以 `Hand` 对象可以在数据库中以 text或者character 的columns的形式储存

字段类是什么？

所有Django的字段（当我们在本文档中提到字段时，我们总是指模型字段，而不是指表单字段）是 `django.db.models.Field的子类` 。Django记录有关字段的大多数信息对于所有字段都是通用的 - 名称，帮助文本，唯一性等。存储所有该信息

由 `Field` 处理。我们将详细了解 `Field` 以后可以做什么；现在，足以说，一切都来自 `Field` ，然后自定义类行为的关键片段。

重要的是要意识到Django字段类不是你的model的属性。模型属性包含普通的Python对象。当创建模型类时，在模型中定义的字段类实际上存储在 `Meta` 类中（此处的具体细节不重要）。这是因为当你只是创建和修改属性时，字段类不是必需的。相反，它们提供了在属性值和存储在数据库中或发送到*serializer*之间进行转换的机制。

创建自己的自定义字段时请记住这一点。您编写的Django `Field` 子类提供了以各种方式在Python实例和数据库/序列化器值之间进行转换的机制（例如，在存储值和使用值之间存在差异）。如果这听起来有点棘手，不要担心 - 这将变得更清楚在下面的例子。只要记住，当你想要一个自定义字段时，你最终会创建两个类：

- 第一类是用户将操作的Python对象。他们将它分配给模型属性，他们将从它读取用于显示的目的，这样的东西。这是我们示例中的 `Hand` 类。
- 第二个类是 `Field` 子类。这是知道如何在其永久存储形式和Python表单之间来回转换你的第一个类的类。

编写字段子类

在规划 `Field` 子类时，请先考虑您的新字段与之最相似的现有 `Field` 类。你可以继承一个现有的Django字段并保存自己一些工作吗？如果没有，你应该继承 `Field` 类，所有类都从其中降序。

初始化您的新字段是一个问题，从您的案例中分离出任何特定于常见参数的参数，并将后者传递到 `Field` 的 `__init__()`

在我们的示例中，我们将调用字段 `HandField` 。（最好调用 `Field` 子类 `<Something>Field` ，因此很容易被识别为 `Field` 子类）。它不像任何现有字段，因此我们将直接从 `Field` 子类化：

```
from django.db import models

class HandField(models.Field):

    description = "A hand of cards (bridge style)"

    def __init__(self, *args, **kwargs):
        kwargs['max_length'] = 104
        super(HandField, self).__init__(*args, **kwargs)
```

我们的 `HandField` 接受大多数标准字段选项（见下面的列表），但我们确保它具有固定长度，因为它只需要持有52个卡片值加上他们的衣服；共104个字符。

注意

许多Django的模型字段接受他们不做任何事情的选项。例如，您可以将 `editable` 和 `auto_now` 同时传递到 `django.db.models.DateField` ，它将忽略 `editable` 参数（ `auto_now` 被设置意味着 `editable=False` ）。在这种情况下不会出现错误。

此行为简化了字段类，因为它们不需要检查不必要的选项。它们只是将所有选项传递给父类，然后不再使用它们。这取决于你是否希望字段对他们选择的选项更加严格，或者使用当前字段的更简单，更宽松的行为。

`Field.__init__()` 方法采用以下参数：

- verbose_name
- `name`
- primary_key(关键字)
- max_length(最大长度)
- unique(唯一性)
- blank(空白)
- null(空值)
- db_index(数据库(创建)索引)
- `rel` ：用于相关字段（如 `ForeignKey` ）。仅供高级使用。
- default(默认值)
- editable(可编辑)
- `serialize` ：如果 `False` ，当模型传递给Django的*serializers*时，字段不会被序列化。默认为 `True` 。
- unique_for_date
- unique_for_month
- unique_for_year
- choices(选择)
- help_text
- db_column
- `db_tablespace` ：仅用于创建索引（如果后端支持*tablespaces*）。您通常可以忽略此选项。
- `auto_created` ： `True` 如果自动创建字段，就像模型继承使用的 `OneToOneField` 。仅供高级使用。

在上面列表中没有解释的所有选项具有与正常Django字段相同的含义。有关示例和详细信息，请参阅*field documentation*。

Field deconstruction(model域 解析)

New in Django 1.7:

`deconstruct()` 是Django 1.7及更高版本中migrations框架的一部分。如果您有来自以前版本的自定义字段，则需要添加此方法才能在迁移中使用它们。

写入 `__init__()` 方法的目的是写入 `deconstruct()` 方法。这个方法告诉Django如何获取一个新字段的实例，并将其减少为序列化形式 - 特别是要传递给 `__init__()` 的参数以重新创建它。

如果您没有在继承域的顶部添加任何额外的选项，则无需编写新的 `deconstruct()` 方法。然而，如果你改变了在 `__init__()` 中传递的参数（就像我们在 `HandField` 中），你需要补充传递的值。

`deconstruct()` 的约定很简单；它返回一个包含四个项目的元组：字段的属性名称，字段类的完整导入路径，位置参数（作为列表）和关键字参数（作为dict）。注意，这不同于自定义类的 `deconstruct()` 方法*for custom classes*

作为一个自定义字段作者，你不需要关心前两个值；基础 `Field` 类具有所有代码以计算字段的属性名称和导入路径。然而，你必须关心位置和关键字参数，因为这些可能是你正在改变的事情。

例如，在我们的 `HandField` 类中，我们总是强制在 `__init__()` 中设置max_length。基础 `Field` 类上的 `deconstruct()` 方法将会看到这个，并尝试在关键字参数中返回它；因此，我们可以从关键字参数中删除它的可读性：

```
from django.db import models

class HandField(models.Field):

    def __init__(self, *args, **kwargs):
        kwargs['max_length'] = 104
        super(HandField, self).__init__(*args, **kwargs)

    def deconstruct(self):
        name, path, args, kwargs = super(HandField, self).decons
truct()
        del kwargs["max_length"]
        return name, path, args, kwargs
```

如果您添加了一个新的关键字参数，则需要编写代码以将其值自动添加到 `kwargs` 中：

```
from django.db import models

class CommaSepField(models.Field):
    "Implements comma-separated storage of lists"

    def __init__(self, separator=",", *args, **kwargs):
        self.separator = separator
        super(CommaSepField, self).__init__(*args, **kwargs)

    def deconstruct(self):
        name, path, args, kwargs = super(CommaSepField, self).de
construct()
        # Only include kwarg if it's not the default
        if self.separator != ",":
            kwargs['separator'] = self.separator
        return name, path, args, kwargs
```

更复杂的示例超出了本文档的范围，但请记住 - 对于Field实例的任何配置， `deconstruct()` 必须返回可以传递到 `__init__` 那个状态。

如果您在 `Field` 超类中为参数设置新的默认值，请特别注意；你想要确保它们总是包含，而不是消失，如果他们采取旧的默认值。

另外，尽量避免返回值作为位置参数；在可能的情况下，返回值作为关键字参数，以实现最大的未来兼容性。当然，如果你改变事物的名字比它们在构造函数的参数列表中的位置更多，你可能更喜欢位置，但是请记住，人们将从序列化版本重构你的领域一段时间（可能是几年）这取决于你的迁移活多久。

您可以通过查看包含字段的迁移来查看解构的结果，您可以通过解构和重建字段来测试单元测试中的解构：

```
name, path, args, kwargs = my_field_instance.deconstruct()
new_instance = MyField(*args, **kwargs)
self.assertEqual(my_field_instance.some_attribute, new_instance.
some_attribute)
```

记录自定义字段

和往常一样，你应该记录你的字段类型，让用户知道它是什么。除了为开发人员提供文档字符串外，您还可以允许管理应用程序的用户通过*django.contrib.admindocs*应用程序查看字段类型的简短说明。为此，只需在自定义字段的 `description` 类属性中提供描述性文本即可。在上面的示例中， `admindocs` 应用程序为 `HandField` 显示的描述将是"A手牌（桥牌）"。

在 `django.contrib.admindocs` 显示中，字段描述由 `field.__dict__` ，允许描述包含字段的参数。例如， `CharField` 的描述是：

```
description = _("String (up to %(max_length)s)")
```

有用的方法

创建 `Field` 子类后，您可能会考虑覆盖几个标准方法，具体取决于字段的行为。下面的方法列表大约是重要性的降序，所以从顶部开始。

自定义数据库类型

假设您创建了一个名为 `mytype` 的PostgreSQL自定义类型。您可以子类化 `Field` 并实现 `db_type()` 方法，如下所示：

```
from django.db import models

class MytypeField(models.Field):
    def db_type(self, connection):
        return 'mytype'
```

一旦您拥有 `MytypeField` ，就可以在任何模型中使用它，就像任何其他 `Field` 类型：

```
class Person(models.Model):
    name = models.CharField(max_length=80)
    something_else = MytypeField()
```

如果您打算构建一个不依赖于数据库的应用程序，则应考虑数据库列类型的差异。例如，PostgreSQL中的日期/时间列类型称为 `timestamp` ，而MySQL中的相同列称为 `datetime` 。在 `db_type()` 方法中处理此问题的最简单的方法是检查 `connection.settings_dict['ENGINE']` 属性。

例如：

```
class MyDateField(models.Field):
    def db_type(self, connection):
        if connection.settings_dict['ENGINE'] == 'django.db.back
ends.mysql':
            return 'datetime'
        else:
            return 'timestamp'
```

当框架为您的应用程序构建 `CREATE TABLE` 语句时，Django会调用 `db_type()` 当构造包含模型字段的 `WHERE` 子句时，也就是当使用诸如 `get()` ， `filter()` 和 `exclude()` 并将模型字段作为参数。它不会在任何其他时间调用，因此它可以执行稍微复杂的代码，例如 `connection.settings_dict` 检查上面的示例。

某些数据库列类型接受参数，例如 `CHAR(25)` ，其中参数 `25` 表示最大列长度。在这种情况下，如果在模型中指定参数，而不是在 `db_type()` 方法中硬编码，则它更灵活。例如，使用 `CharMaxlength25Field` 没有什么意义，如下所示：

```
# This is a silly example of hard-coded parameters.
class CharMaxlength25Field(models.Field):
    def db_type(self, connection):
        return 'char(25)'

# In the model:
class MyModel(models.Model):
    # ...
    my_field = CharMaxlength25Field()
```

这样做的更好的方法是使参数在运行时可指定 - 即当类被实例化时。要这样做，只需实现 `Field.__init__()` ，像这样：

```
# This is a much more flexible example.
class BetterCharField(models.Field):
    def __init__(self, max_length, *args, **kwargs):
        self.max_length = max_length
        super(BetterCharField, self).__init__(*args, **kwargs)

    def db_type(self, connection):
        return 'char(%s)' % self.max_length

# In the model:
class MyModel(models.Model):
    # ...
    my_field = BetterCharField(25)
```

最后，如果您的列需要真正复杂的SQL设置，请从 `db_type()` 返回 `None` 。这将导致Django的SQL创建代码跳过此字段。然后，您将负责以某种其他方式在正确的表中创建列，当然，但是这让你有一种方法来告诉Django脱离方式。

将值转换为Python对象

Changed in Django 1.8:

历史上，Django提供了一个称为 `SubfieldBase` 的元类，它在赋值时总是调用 `to_python()` 。这对于自定义数据库转换，聚合或值查询没有很好地发挥作用，因此已被替换为 `from_db_value()` 。

如果您的自定义 `Field` 类处理比字符串，日期，整数或浮点数更复杂的数据结构，则可能需要覆盖 `from_db_value()` 和 `to_python()` 。

如果存在于字段子类，则在从数据库加载数据（包括在聚合和 `values()` 调用）的所有情况下将调用 `from_db_value()` 。

`to_python()` 通过反序列化和从表单使用的 `clean()` 方法调用。

作为一般规则， `to_python()` 应优雅地处理以下任何参数：

- 正确类型的实例（例如，在我们正在进行的示例中， `Hand` ）。
- 字符串
- `None` （如果字段允许 `null=True` ）

在我们 `HandField` 类种,我们使用 VARCHAR 域在数据库中存储我们的数据, 因此我们需要在 `from_db_value()` 函数中有能力处理字符串跟 `None` 值。
在 `to_python()` 中，我们还需要处理 `Hand` 实例：

```
import re

from django.core.exceptions import ValidationError
from django.db import models

def parse_hand(hand_string):
    """Takes a string of cards and splits into a full hand."""
    p1 = re.compile('.{26}')
    p2 = re.compile('..')
    args = [p2.findall(x) for x in p1.findall(hand_string)]
    if len(args) != 4:
        raise ValidationError("Invalid input for a Hand instance
")
    return Hand(*args)

class HandField(models.Field):
    # ...

    def from_db_value(self, value, expression, connection, conte
xt):
        if value is None:
            return value
        return parse_hand(value)

    def to_python(self, value):
        if isinstance(value, Hand):
            return value

        if value is None:
            return value

        return parse_hand(value)
```

注意，我们总是从这些方法返回 `Hand` 实例。这是我们要存储在模型属性中的Python对象类型。

对于 `to_python()` ，如果在值转换期间出现任何错误，您应该引发一个 `ValidationError` 异常。

将Python对象转换为查询值

由于使用数据库需要以两种方式进行转换，因此如果覆盖 `to_python()`，您还必须重写 `get_prep_value()` 将Python对象转换回查询值。

例如：

```
class HandField(models.Field):
    # ...

    def get_prep_value(self, value):
        return ''.join([''.join(l) for l in (value.north,
                value.east, value.south, value.west)])
```

警告

如果您的自定义字段使用MySQL的 `CHAR`，`VARCHAR` 或 `TEXT` 类型，则必须确保 `get_prep_value()` 当对这些类型执行查询并且提供的值是整数时，MySQL执行灵活和意外匹配，这可能导致查询在其结果中包含意外的对象。如果您始终从 `get_prep_value()` 返回字符串类型，则不会发生此问题。

将查询值转换为数据库值

某些数据类型（例如，日期）需要采用特定格式，才能供数据库后端使用。`get_db_prep_value()` 是应进行这些转换的方法。将用于查询的特定连接作为 `connection` 参数传递。这允许您使用后端特定的转换逻辑（如果需要）。

例如，Django对其 `BinaryField` 使用以下方法：

```
def get_db_prep_value(self, value, connection, prepared=False):
    value = super(BinaryField, self).get_db_prep_value(value, co
nnection, prepared)
    if value is not None:
        return connection.Database.Binary(value)
    return value
```

如果您的自定义字段在保存时需要进行特殊转换，但与用于常规查询参数的转换不同，则可以覆盖 `get_db_prep_save()`。

保存前预处理值

如果要在保存之前预处理值，可以使用 `pre_save()`。例如，在 `auto_now` 或 `auto_now_add` 的情况下，Django的 `DateTimeField` 使用此方法正确设置属性。

如果您覆盖此方法，则必须在结尾返回属性的值。如果对值进行任何更改，您还应该更新模型的属性，以便保存对模型的引用的代码将始终看到正确的值。

准备用于数据库查找的值

与值转换一样，为数据库查找准备一个值是一个两阶段过程。

`get_prep_lookup()` 执行查找准备的第一阶段：类型转换和数据验证。

在查找（在SQL中使用 `WHERE` 约束）时，准备传递到数据库的 `value` 。The `lookup_type` parameter will be one of the valid Django filter lookups: `exact` , `iexact` , `contains` , `icontains` , `gt` , `gte` , `lt` , `lte` , `in` , `startswith` , `istartswith` , `endswith` , `iendswith` , `range` , `year` , `month` , `day` , `isnull` , `search` , `regex` , and `iregex` .

New in Django 1.7:

如果您使用*Custom lookups*，则 `lookup_type` 可以是项目自定义查找使用的任何 `lookup_name` 。

您的方法必须准备好处理所有这些 `lookup_type` 值，如果 `value` 的排序错误，则应提出 `ValueError` 当你期望一个对象，例如）或一个 `TypeError` 如果你的字段不支持这种类型的查找。对于许多字段，您可以通过处理需要对字段进行特殊处理的查找类型，并将剩余部分传递给父类的 `get_db_prep_lookup()` 方法。

如果你需要实现 `get_db_prep_save()` ，你通常需要实现 `get_prep_lookup()` 。If you don't, `get_prep_value()` will be called by the default implementation, to manage `exact` , `gt` , `gte` , `lt` , `lte` , `in` and `range` lookups.

您可能还希望实现此方法以限制可以与自定义字段类型一起使用的查找类型。

注意，对于 `"range"` 和 `"in"` 查找中， `get_prep_lookup` 将接收一个对象列表需要将它们转换为适当类型的事物列表以传递到数据库。大多数时候，你可以重用 `get_prep_value()` ，或者至少考虑一些常见的部分。

例如，以下代码实现 `get_prep_lookup` 以将接受的查找类型限制为 `exact` 和 `in` ：

```
class HandField(models.Field):
    # ...

    def get_prep_lookup(self, lookup_type, value):
        # We only handle 'exact' and 'in'. All others are errors
.
        if lookup_type == 'exact':
            return self.get_prep_value(value)
        elif lookup_type == 'in':
            return [self.get_prep_value(v) for v in value]
        else:
            raise TypeError('Lookup type %r not supported.' % lo
okup_type)
```

要执行查找所需的特定于数据库的数据转换，您可以覆盖 `get_db_prep_lookup()` 。

指定模型字段的表单字段

要自定义 `ModelForm` 使用的表单字段，您可以覆盖 `formfield()` 。

表单字段类可以通过 `form_class` 和 `choices_form_class` 参数指定；如果字段具有指定的选项，则使用后者，否则。如果不提供这些参数，将使用 `CharField` 或 `TypedChoiceField` 。

所有 `kwargs` 字典都直接传递到表单字段的 `__init__()` 方法。通常，你需要做的是为 `form_class` （也许 `choices_form_class` ）参数设置好的默认值，然后委托进一步处理到父类。这可能需要您编写自定义表单字段（甚至是窗体小部件）。有关此信息，请参阅*forms documentation*。

继续我们正在进行的示例，我们可以将 `formfield()` 方法写为：

```
class HandField(models.Field):
    # ...

    def formfield(self, **kwargs):
        # This is a fairly standard way to set up some defaults
        # while letting the caller override them.
        defaults = {'form_class': MyFormField}
        defaults.update(kwargs)
        return super(HandField, self).formfield(**defaults)
```

这假设我们已经导入了一个 `MyFormField` 字段类（它有自己的默认小部件）。本文档不包括编写自定义表单字段的详细信息。

仿真内置字段类型

如果您创建了 `db_type()` 方法，则不需要担心 `get_internal_type()` - 它不会被使用太多。但有时，您的数据库存储在类型上与其他字段类似，因此您可以使用其他字段的逻辑来创建正确的列。

例如：

```
class HandField(models.Field):
    # ...

    def get_internal_type(self):
        return 'CharField'
```

不管我们使用哪个数据库后端，这将意味着 `migrate` 和其他SQL命令创建用于存储字符串的正确的列类型。

如果 `get_internal_type()` 返回Django对于您正在使用的数据库后端不为已知的字符串，也就是说，它不会出现在 django.db.backends.<db_name>.base. DatabaseWrapper.data_types - 字符串仍然被序列化器使用，但是默认的 `db_type()` 方法将返回 `None` 。有关这可能有用的原因，请参见 `db_type()` 的文档。如果你要在Django之外的其他地方使用serializer输出，那么将描述性字符串作为序列化字段的字段类型是一个有用的想法。

转换字段数据以进行序列化

要自定义序列化程序如何序列化值，您可以覆盖 `value_to_string()` 。调用 `Field._get_val_from_obj(obj)` 是获取值序列化的最佳方式。例如，由于我们的 `HandField` 使用字符串作为其数据存储，我们可以重用一些现有的转换代码：

```
class HandField(models.Field):
    # ...

    def value_to_string(self, obj):
        value = self._get_val_from_obj(obj)
        return self.get_prep_value(value)
```

一些一般建议

编写自定义字段可能是一个棘手的过程，特别是如果您在Python类型与数据库和序列化格式之间进行复杂的转换。这里有几个提示，使事情进行得更顺利：

1. 查看现有的Django字段（在 `django/db/models/fields/__init__.py` ）获取灵感。尝试找到一个类似于你想要的字段，并扩展它一点，而不是从头创建一个全新的字段。
2. 将一个 `__str__()` （ `__unicode__()` 在Python 2）方法放在类中作为一个字段。有很多地方的字段代码的默认行为是调用 `force_text()` 上的值。（在本文档的示例中， `value` 将是 `Hand` 实例，而不是 `HandField` ）。So if your `__str__()` method (`__unicode__()` on Python 2) automatically converts to the string form of your Python object, you can save yourself a lot of work.

写一个 `FileField`

除了上述方法，处理文件的字段还有一些其他特殊要求，必须考虑到。 `FileField` 提供的大多数机制（例如控制数据库存储和检索）可以保持不变，使子类处理支持特定类型文件的挑战。

Django提供了一个 `File` 类，用作文件内容和操作的代理。这可以被子类化以自定义如何访问文件，以及可用的方法。它位于 `django.db.models.fields.files` ，其默认行为在*file documentation*中解释。

一旦创建了 `File` 的子类，就必须告诉新的 `FileField` 子类使用它。为此，只需将新的 `File` 子类分配给 `FileField` 子类的特殊 `attr_class` 属性。

几个建议

除了上面的细节，还有一些指南可以大大提高字段代码的效率和可读性。

1. Django自己的 `ImageField` （在 `django/db/models/fields/files.py` ）的源代码是一个很好的例子来说明如何子类化 `FileField` 以支持特定类型的文件，因为其包含上述所有技术。
2. 尽可能缓存文件属性。由于文件可以存储在远程存储系统中，因此检索它们可能花费额外的时间或甚至金钱，这并不总是必需的。一旦检索到文件以获得关于其内容的一些数据，就尽可能地缓存那些数据，以减少在该信息的后续调用中必须检索文件的次数。

多数据库

这篇主题描述Django 对多个数据库的支持。大部分Django 文档假设你只和一个数据库打交道。如果你想与多个数据库打交道，你将需要一些额外的步骤。

定义你的数据库

在Django中使用多个数据库的第一步是告诉Django 你将要使用的数据库服务器。这通过使用 `DATABASES` 设置完成。该设置映射数据库别名到一个数据库连接设置的字典，这是整个Django 中引用一个数据库的方式。字典中的设置在 `DATABASES` 文档中有完整描述。

你可以为数据库选择任何别名。然而， `default` 这个别名具有特殊的含义。当没有选择其它数据库时，Django 使用具有 `default` 别名的数据库。

下面是 `settings.py` 的一个示例片段，它定义两个数据库 —— 一个默认的PostgreSQL 数据库和一个叫做 `users` 的MySQL 数据库：

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'USER': 'postgres_user',
        'PASSWORD': 's3krit'
    },
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'priv4te'
    }
}
```

如果 `default` 数据库在你的项目中不合适，你需要小心地永远指定是想使用的数据库。Django 要求 `default` 数据库必须定义，但是其参数字典可以保留为空如果不使用它。若要这样做，你必须为你的所有的应用的模型建立 `DATABASE_ROUTERS` ，包括正在使用的 `contrib` 中的应用和第三方应用，以使得不会有查询被路由到默认的数据库。下面是 `settings.py` 的一个示例片段，它定义两个非默认的数据库，其中 `default` 有意保留为空：

```
DATABASES = {
    'default': {},
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'superS3cret'
    },
    'customers': {
        'NAME': 'customer_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_cust',
        'PASSWORD': 'veryPriv@ate'
    }
}
```

如果你试图访问在 `DATABASES` 设置中没有定义的数据库，Django 将抛出一个 `django.db.utils.ConnectionDoesNotExist` 异常。

同步你的数据库

`migrate` 管理命令一次操作一个数据库。默认情况下，它在 `default` 数据库上操作，但是通过提供一个 `--database` 参数，你可以告诉 `migrate` 同步一个不同的数据库。因此，为了同步所有模型到我们示例中的所有数据库，你将需要调用：

```
$ ./manage.py migrate
$ ./manage.py migrate --database=users
```

如果你不想每个应用都被同步到同一台数据库上，你可以定义一个数据库路由，它实现一个策略来控制特定模型的访问性。

使用其它管理命令

其它 `django-admin` 命令与数据库交互的方式与 `migrate` 相同 —— 它们都一次只操作一个数据库，并使用 `--database` 来控制使用的数据库。

数据库自动路由

使用多数据库最简单的方法是建立一个数据库路由模式。默认的路由模式确保对象'粘滞'在它们原始的数据库上（例如，从 `foo` 数据库中获取的对象将保存在同一个数据库中）。默认的路由模式还确保如果没有指明数据库，所有的查询都回归到 `default` 数据库中。

你不需要做任何事情来激活默认的路由模式 —— 它在每个Django项目上'直接'提供。然而，如果你想实现更有趣的数据库分配行为，你可以定义并安装你自己的数据库路由。

数据库路由

数据库路由是一个类，它提供4个方法：

`db_for_read(model, **hints)`

建议model类型的对象的读操作应该使用的数据库。

如果一个数据库操作能够提供其它额外的信息可以帮助选择一个数据库，它将在 `hints` 字典中提供。合法的 `hints` 的详细信息在下文给出。

如果没有建议，则返回 `None` 。

`db_for_write(model, **hints)`

建议Model 类型的对象的写操作应该使用的数据库。

如果一个数据库操作能够提供其它额外的信息可以帮助选择一个数据库，它将在 `hints` 字典中提供。 合法的 `hints` 的详细信息在下文给出。

如果没有建议，则返回None。

`allow_relation(obj1, obj2, **hints)`

如果 `obj1` 和 `obj2` 之间应该允许关联则返回 `True` ，如果应该防止关联则返回 `False` ，如果路由无法判断则返回 `None` 。这是纯粹的验证操作，外键和多对多操作使用它来决定两个对象之间是否应该允许一个关联。

`allow_migrate(db, app_label, model_name=None, **hints)`

定义迁移操作是否允许在别名为 `db` 的数据库上运行。如果操作应该运行则返回 `True` ，如果不应该运行则返回 `False` ，如果路由无法判断则返回 `None` 。

位置参数 `app_label` 是正在迁移的应用的标签。

大部分迁移操作设置 `model_name` 的值为正在迁移的模型的 `model._meta.model_name` （模型的 `__name__` 的小写）。对于RunPython和RunSQL 操作它的值为 `None` ，除非这两个操作使用hint 提供它。

`hints` 用于某些操作来传递额外的信息给路由。

当设置了 `model_name` 时， `hints` 通常通过键' `model` '包含该模型的类。注意，它可能是一个历史模型，因此不会有自定的属性、方法或管理器。你应该只依赖 `_meta` 。

这个方法还可以用来决定一个给定数据库上某个模型的可用性。

注意，如果这个方法返回 `False` ，迁移将默默地不会在模型上做任何操作。这可能导致你应用某些操作之后出现损坏的外键、表多余或者缺失。

```
Changed in Django 1.8:

The signature of allow_migrate has changed significantly from pr
evious versions. See the deprecation notes for more details.
```

路由不必提供所有这些方法 —— 它可以省略一个或多个。如果某个方法缺失，在做相应的检查时Django 将忽略该路由。

Hints

`Hint` 由数据库路由接收，用于决定哪个数据库应该接收一个给定的请求。

目前，唯一一个提供的hint 是instance，它是一个对象实例，与正在进行的读或者写操作关联。This might be the instance that is being saved, or it might be an instance that is being added in a many-to-many relation. In some cases, no instance hint will be provided at all. The router checks for the existence of an instance hint, and determine if that hint should be used to alter routing behavior.

使用路由

数据库路由使用 `DATABASE_ROUTERS` 设置安装。这个设置定义一个类名的列表，其中每个类表示一个路由，它们将被主路由（ `django.db.router` ）使用。

Django 的数据库操作使用主路由来分配数据库的使用。每当一个查询需要知道使用哪一个数据库时，它将调用主路由，并提供一个模型和一个 `Hint` （可选）。Django 然后依次测试每个路由直至找到一个数据库的建议。如果找不到建议，它将尝试 `Hint` 实例的当前 `_state.db` 。如果没有提供Hint 实例，或者该实例当前没有数据库状态，主路由将分配 `default` 数据库。

一个例子

> 只是为了示例！
>
> 这个例子的目的是演示如何使用路由这个基本结构来改变数据库的使用。它有意忽略一些复杂的问题，目的是为了演示如何使用路由。
>
> 如果 `myapp` 中的任何一个模型包含与其它 数据库之外的模型的关联，这个例子将不能工作。跨数据的关联引入引用完整性问题，Django目前还无法处理。
>
> `Primary/replica` （在某些数据库中叫做 `master/slave` ）配置也是有缺陷的 —— 它不提供任何处理Replication lag 的解决办法（例如，因为写入同步到replica 需要一定的时间，这会引入查询的不一致）。It also doesn't consider the interaction of transactions with the database utilization strategy.

那么 —— 在实际应用中这以为着什么？让我们看一下另外一个配置的例子。这个配置将有几个数据库：一个用于 `auth` 应用，所有其它应用使用一个具有两个读 `replica` 的 `primary/replica` 。下面是表示这些数据库的设置：

```
DATABASES = {
    'auth_db': {
        'NAME': 'auth_db',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'swordfish',
    },
    'primary': {
        'NAME': 'primary',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'spam',
    },
    'replica1': {
        'NAME': 'replica1',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'eggs',
    },
    'replica2': {
        'NAME': 'replica2',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'bacon',
    },
}
```

现在我们将需要处理路由。首先，我们需要一个路由，它知道发送 `auth` 应用的查询到 `auth_db` ：

```
class AuthRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None

    def db_for_write(self, model, **hints):
        """
        Attempts to write auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None

    def allow_relation(self, obj1, obj2, **hints):
        """
        Allow relations if a model in the auth app is involved.
        """
        if obj1._meta.app_label == 'auth' or \
           obj2._meta.app_label == 'auth':
           return True
        return None

    def allow_migrate(self, db, app_label, model=None, **hints):
        """
        Make sure the auth app only appears in the 'auth_db'
        database.
        """
        if app_label == 'auth':
            return db == 'auth_db'
        return None
```

我们还需要一个路由，它发送所有其它应用的查询到 `primary/replica` 配置，并随机选择一个 `replica` 来读取：

```
import random

class PrimaryReplicaRouter(object):
    def db_for_read(self, model, **hints):
        """
        Reads go to a randomly-chosen replica.
        """
        return random.choice(['replica1', 'replica2'])

    def db_for_write(self, model, **hints):
        """
        Writes always go to primary.
        """
        return 'primary'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects ar
e
        in the primary/replica pool.
        """
        db_list = ('primary', 'replica1', 'replica2')
        if obj1._state.db in db_list and obj2._state.db in db_li
st:
            return True
        return None

    def allow_migrate(self, db, app_label, model=None, **hints):
        """
        All non-auth models end up in this pool.
        """
        return True
```

最后，在设置文件中，我们添加如下内容（替换 `path.to.` 为该路由定义所在的真正路径）：

```
DATABASE_ROUTERS = ['path.to.AuthRouter', 'path.to.PrimaryReplic
aRouter']
```

路由处理的顺序非常重要。路由的查询将按照DATABASE_ROUTERS设置中列出的顺序进行。在这个例子中， `AuthRouter` 在 `PrimaryReplicaRouter` 之前处理，因此 `auth` 中的模型的查询处理在其它模型之前。如果 `DATABASE_ROUTERS` 设置按其它顺序列出这两个路由， `PrimaryReplicaRouter.allow_migrate()` 将先处理。 `PrimaryReplicaRouter` 中实现的捕获所有的查询，这意味着所有的模型可以位于所有的数据库中。

建立这个配置后，让我们运行一些Django 代码：

```
>>> # This retrieval will be performed on the 'auth_db' database
>>> fred = User.objects.get(username='fred')
>>> fred.first_name = 'Frederick'

>>> # This save will also be directed to 'auth_db'
>>> fred.save()

>>> # These retrieval will be randomly allocated to a replica da
tabase
>>> dna = Person.objects.get(name='Douglas Adams')

>>> # A new object has no database allocation when created
>>> mh = Book(title='Mostly Harmless')

>>> # This assignment will consult the router, and set mh onto
>>> # the same database as the author object
>>> mh.author = dna

>>> # This save will force the 'mh' instance onto the primary da
tabase...
>>> mh.save()

>>> # ... but if we re-retrieve the object, it will come back on
 a replica
>>> mh = Book.objects.get(title='Mostly Harmless')
```

手动选择一个数据库

Django 还提供一个API，允许你在你的代码中完全控制数据库的使用。人工指定的数据库的优先级高于路由分配的数据库。

为**QuerySet**手动选择一个数据库

你可以在 `QuerySet` “链”的任意节点上为 `QuerySet` 选择数据库 。只需要 `在QuerySet` 上调用 `using()` 就可以让 `QuerySet` 使用一个指定的数据库。

`using()` 接收单个参数：你的查询想要运行的数据库的别名。例如：

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()

>>> # So will this.
>>> Author.objects.using('default').all()

>>> # This will run on the 'other' database.
>>> Author.objects.using('other').all()
```

为save() 选择一个数据库

对 `Model.save()` 使用 `using` 关键字来指定数据应该保存在哪个数据库。

例如，若要保存一个对象到legacy_users 数据库，你应该使用：

```
>>> my_object.save(using='legacy_users')
```

如果你不指定 `using` ，`save()` 方法将保存到路由分配的默认数据库中。

将对象从一个数据库移动到另一个数据库

如果你已经保存一个实例到一个数据库中，你可能很想使用 `save(using=...)` 来迁移该实例到一个新的数据库中。然而，如果你不使用正确的步骤，这可能导致意外的结果。

考虑下面的例子：

```
>>> p = Person(name='Fred')
>>> p.save(using='first')  # (statement 1)
>>> p.save(using='second') # (statement 2)
```

在statement 1中，一个新的 `Person` 对象被保存到 `first` 数据库中。此时 `p` 没有主键，所以Django 发出一个 `SQL INSERT` 语句。这会创建一个主键，且Django 将此主键赋值给 `p` 。

当保存在statement 2中发生时， `p` 已经具有一个主键，Django 将尝试在新的数据库上使用该主键。如果该主键值在 `second` 数据库中没有使用，那么你不会遇到问题 —— 该对象将被复制到新的数据库中。

然而，如果 `p` 的主键在 `second` 数据库上已经在使用 `second` 数据库中的已经存在的对象将在 `p` 保存时被覆盖。

你可以用两种方法避免这种情况。首先，你可以清除实例的主键。如果一个对象没有主键，Django 将把它当做一个新的对象，这将避免 `second` 数据库上数据的丢失：

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.pk = None # Clear the primary key.
>>> p.save(using='second') # Write a completely new object.
```

第二种方法是使用 `force_insert` 选项来 `save()` 以确保Django 使用一个 `INSERT SQL` ：

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.save(using='second', force_insert=True)
```

这将确保名称为 `Fred` 的 `Person` 在两个数据库上具有相同的主键。在你试图保存到 `second` 数据库，如果主键已经在使用，将会引抛出发一个错误。

选择一个数据库用于删除表单

默认情况下，删除一个已存在对象的调用将在与获取对象时使用的相同数据库上执行：

```
>>> u = User.objects.using('legacy_users').get(username='fred')
>>> u.delete() # will delete from the `legacy_users` database
```

要指定删除一个模型时使用的数据库，可以对 `Model.delete()` 方法使用 `using` 关键字参数。这个参数的工作方式与 `save()` 的 `using` 关键字参数一样。

例如，你正在从 `legacy_users` 数据库到 `new_users` 数据库迁移一个 `User` ，你可以使用这些命令：

```
>>> user_obj.save(using='new_users')
>>> user_obj.delete(using='legacy_users')
```

多个数据库上使用管理器

在管理器上使用 `db_manager()` 方法来让管理器访问非默认的数据库。

例如，你有一个自定义的管理器方法，它访问数据库时候用——— `User.objects.create_user()` 。因为 `create_user()` 是一个管理器方法，不是一个 `QuerySet` 方法，你不可以使用 `User.objects.using('new_users').create_user()` 。（ `create_user()` 方法只能在 `User.objects` 上使用，而不能在从管理器得到的 `QuerySet` 上使用）。解决办法是使用 `db_manager()` ，像这样：

```
User.objects.db_manager('new_users').create_user(...)
```

`db_manager()` 返回一个绑定在你指定的数据上的一个管理器。

多数据库上使用**get_queryset()**

如果你正在覆盖你的管理器上的 `get_queryset()` ，请确保在其父类上调用方法（使用 `super()` ）或者正确处理管理器上的 `_db` 属性（一个包含将要使用的数据库名称的字符串）。

例如，如果你想从 `get_queryset` 方法返回一个自定义的 `QuerySet` 类，你可以这样做：

```
class MyManager(models.Manager):
    def get_queryset(self):
        qs = CustomQuerySet(self.model)
        if self._db is not None:
            qs = qs.using(self._db)
        return qs
```

Django 的管理站点中使用多数据库

Django 的管理站点没有对多数据库的任何显式的支持。如果你给数据库上某个模型提供的管理站点不想通过你的路由链指定，你将需要编写自定义的 `ModelAdmin` 类用来将管理站点导向一个特殊的数据库。

`ModelAdmin` 对象具有5个方法，它们需要定制以支持多数据库：

```
class MultiDBModelAdmin(admin.ModelAdmin):
    # A handy constant for the name of the alternate database.
    using = 'other'

    def save_model(self, request, obj, form, change):
        # Tell Django to save objects to the 'other' database.
        obj.save(using=self.using)

    def delete_model(self, request, obj):
        # Tell Django to delete objects from the 'other' databas
e
        obj.delete(using=self.using)

    def get_queryset(self, request):
        # Tell Django to look for objects on the 'other' databas
e.
        return super(MultiDBModelAdmin, self).get_queryset(reque
st).using(self.using)

    def formfield_for_foreignkey(self, db_field, request=None, *
*kwargs):
        # Tell Django to populate ForeignKey widgets using a que
ry
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_fore
ignkey(db_field, request=request, using=self.using, **kwargs)

    def formfield_for_manytomany(self, db_field, request=None, *
*kwargs):
        # Tell Django to populate ManyToMany widgets using a que
ry
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_many
tomany(db_field, request=request, using=self.using, **kwargs)
```

这里提供的实现实现了一个多数据库策略，其中一个给定类型的所有对象都将保存在一个特定的数据库上（例如，所有的 `User` 保存在 `other` 数据库中）。如果你的多数据库的用法更加复杂，你的 `ModelAdmin` 将需要反映相应的策略。

`Inlines` 可以用相似的方式处理。它们需要3个自定义的方法：

```
class MultiDBTabularInline(admin.TabularInline):
    using = 'other'

    def get_queryset(self, request):
        # Tell Django to look for inline objects on the 'other'
database.
        return super(MultiDBTabularInline, self).get_queryset(re
quest).using(self.using)

    def formfield_for_foreignkey(self, db_field, request=None, *
*kwargs):
        # Tell Django to populate ForeignKey widgets using a que
ry
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_f
oreignkey(db_field, request=request, using=self.using, **kwargs)

    def formfield_for_manytomany(self, db_field, request=None, *
*kwargs):
        # Tell Django to populate ManyToMany widgets using a que
ry
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_m
anytomany(db_field, request=request, using=self.using, **kwargs)
```

一旦你写好你的模型管理站点的定义，它们就可以使用任何 `Admin` 实例来注册：

```
from django.contrib import admin

# Specialize the multi-db admin objects for use with specific mo
dels.
class BookInline(MultiDBTabularInline):
    model = Book

class PublisherAdmin(MultiDBModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, MultiDBModelAdmin)
admin.site.register(Publisher, PublisherAdmin)

othersite = admin.AdminSite('othersite')
othersite.register(Publisher, MultiDBModelAdmin)
```

这个例子建立两个管理站点。在第一个站点上， `Author` 和 `Publisher` 对象被暴露出来； `Publisher` 对象具有一个表格的内联，显示该出版社出版的书籍。第二个站点只暴露 `Publishers` ，而没有内联。

多数据库上使用原始游标

如果你正在使用多个数据库，你可以使用 `django.db.connections` 来获取特定数据库的连接（和游标）：`django.db.connections` 是一个类字典对象，它允许你使用别名来获取一个特定的连接：

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
```

多数据库的局限

跨数据库关联

Django 目前不提供跨多个数据库的外键或多对多关系的支持。如果你使用一个路由来路由分离到不同的数据库上，这些模型定义的任何外键和多对多关联必须在单个数据库的内部。

这是因为引用完整性的原因。为了保持两个对象之间的关联，Django 需要知道关联对象的主键是合法的。如果主键存储在另外一个数据库上，判断一个主键的合法性不是很容易。

如果你正在使用Postgres、Oracle或者MySQ 的InnoDB，这是数据库完整性级别的强制要求 —— 数据库级别的主键约束防止创建不能验证合法性的关联。

然而，如果你正在使用SQLite 或MySQL的MyISAM 表，则没有强制性的引用完整性；结果是你可以'伪造'跨数据库的外键。但是Django 官方不支持这种配置。

Contrib 应用的行为

有几个Contrib 应用包含模型，其中一些应用相互依赖。因为跨数据库的关联是不可能的，这对你如何在数据库之间划分这些模型带来一些限制：

- `contenttypes.ContentType` 、 `sessions.Session` 和 `sites.Site` 可以存储在分开存储在不同的数据库中，只要给出合适的路由
- `auth` 模型 —— `User` 、 `Group` 和 `Permission` —— 关联在一起并与 `ContentType` 关联，所以它们必须与 `ContentType` 存储在相同的数据库中。
- `admin` 依赖 `auth` ，所以它们的模型必须与 `auth` 在同一个数据库中。
- `flatpages` 和 `redirects` 依赖 `sites` ，所以它们必须与 `sites` 在同一个数据库中。

另外，一些对象在migrate在数据库中创建一张表后自动创建：

- 一个默认的 `Site` ，
- 为每个模型创建一个 `ContentType` （包括没有存储在同一个数据库中的模型），
- 为每个模型创建3个 `Permission` （包括不是存储在同一个数据库中的模型）。

对于常见的多数据库架构，将这些对象放在多个数据库中没有什么用处。常见的数据库架构包括 `primary/replica` 和连接到外部的数据库。因此，建议写一个数据库路由，它只允许同步这3个模型到一个数据中。对于不需要将表放在多个数据库中的Contrib 应用和第三方应用，可以使用同样的方法。

> 警告
>
> 如果你将 `Content Types` 同步到多个数据库中，注意它们的主键在数据库之间可能不一致。这可能导致数据损坏或数据丢失。

> 译者：Django 文档协作翻译小组，原文：Multiple databases。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

自定义查找

```
New in Django 1.7.
```

Django为过滤提供了大量的内建的查找（例如， `exact` 和 `icontains` ）。这篇文档阐述了如何编写自定义查找，以及如何修改现存查找的功能。关于查找的API参考，详见查找API参考。

一个简单的查找示例

让我们从一个简单的自定义查找开始。我们会编写一个自定义查找 `ne` ，提供和 `exact` 相反的功能。 `Author.objects.filter(name__ne='Jack')` 会转换成下面的SQL：

```
"author"."name" <> 'Jack'
```

这条SQL是后端独立的，所以我们并不需要担心不同的数据库。

实现它需要两个步骤。首先我们需要实现这个查找，然后我们需要告诉Django它的信息。实现是十分简单直接的：

```
from django.db.models import Lookup

class NotEqual(Lookup):
    lookup_name = 'ne'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s <> %s' % (lhs, rhs), params
```

我们只需要在我们想让查找应用的字段上调用 `register_lookup` ，来注册 `NotEqual` 查找。这种情况下，查找在所有 `Field` 的子类都起作用，所以我们直接使用 `Field` 注册它。

```
from django.db.models.fields import Field
Field.register_lookup(NotEqual)
```

也可以使用装饰器模式来注册查找：

```
from django.db.models.fields import Field

@Field.register_lookup
class NotEqualLookup(Lookup):
    # ...
```

```
Changed in Django 1.8:

新增了使用装饰器模式的能力。
```

我们现在可以为任何 `foo` 字段使用 `foo__ne` 。你需要确保在你尝试创建使用它的任何查询集之前完成注册。你应该把实现放在 `models.py` 文件中，或者在 `AppConfig` 的 `ready()` 方法中注册查找。

现在让我们深入观察这个实现，首先需要的属性是 `lookup_name` 。这需要让ORM理解如何去解释 `name__ne` ，以及如何使用 `NotEqual` 来生成SQL。按照惯例，这些名字一般是只包含字母的小写字符串，但是唯一硬性的要求是不能够包含字符串 `__` 。

然后我们需要定义 `as_sql` 方法。这个方法需要传入一个 `SQLCompiler` 对象，叫做 `compiler` ，以及活动的数据库连接。 `SQLCompiler` 对象并没有记录，但是我们需要知道的唯一一件事就是他们拥有 `compile()` 方法，这个方法返回一个元组，含有SQL字符串和要向字符串插入的参数。在多数情况下，你并不需要世界使用它，并且可以把它传递给 `process_lhs()` 和 `process_rhs()` 。

`Lookup` 作用于两个值，lhs和rhs，分别是左边和右边。左边的值一般是个字段的引用，但是它可以是任何实现了查询表达式API的对象。右边的值由用户提供。在例子 `Author.objects.filter(name__ne='Jack')` 中，左边的值是 `Author` 模型的 `name` 字段的引用，右边的值是 `'Jack'` 。

我们可以调用 `process_lhs` 和 `process_rhs` 来将它们转换为我们需要的SQL值，使用之前我们描述的 `compiler` 对象。

最后我们用 `<>` 将这些部分组合成SQL表达式，然后将所有参数用在查询中。然后我们返回一个元组，包含生成的SQL字符串以及参数。

一个简单的转换器示例

上面的自定义转换器是极好的，但是一些情况下你可能想要把查找放在一起。例如，假设我们构建一个应用，想要利用 `abs()` 操作符。我们有用一个 `Experiment` 模型，它记录了起始值，终止值，以及变化量（起始值 - 终止值）。我们想要寻找所有变化量等于一个特定值的实验（ `Experiment.objects.filter(change__abs=27)` ），或者没有达到指定值的实验（ `Experiment.objects.filter(change__abs__lt=27)` ）。

> 注意
>
> 这个例子一定程度上很不自然，但是很好地展示了数据库后端独立的功能范围，并且没有重复实现Django中已有的功能。

我们从编写 `AbsoluteValue` 转换器来开始。这会用到SQL函数 `ABS()` ，来在比较之前转换值。

```
from django.db.models import Transform

class AbsoluteValue(Transform):
    lookup_name = 'abs'

    def as_sql(self, compiler, connection):
        lhs, params = compiler.compile(self.lhs)
        return "ABS(%s)" % lhs, params
```

接下来，为 `IntegerField` 注册它：

```
from django.db.models import IntegerField
IntegerField.register_lookup(AbsoluteValue)
```

我们现在可以执行之前的查询。 `Experiment.objects.filter(change__abs=27)` 会生成下面的SQL：

```
SELECT ... WHERE ABS("experiments"."change") = 27
```

通过使用 `Transform` 来替代 `Lookup` ，这说明了我们能够把以后更多的查找放到一起。所以 `Experiment.objects.filter(change__abs__lt=27)` 会生成以下的SQL：

```
SELECT ... WHERE ABS("experiments"."change") < 27
```

注意在没有指定其他查找的情况中，Django会将 `change__abs=27` 解释为 `change__abs__exact=27` 。

当寻找在 `Transform` 之后，哪个查找可以使用的时候，Django使用 `output_field` 属性。因为它并没有修改，我们在这里并不指定，但是假设我们在一些字段上应用AbsoluteValue，这些字段代表了一个更复杂的类型（比如说与原点（origin）相关的一个点，或者一个复数（complex number））。之后我们可能想指定，转换要为进一步的查找返回 `FloatField` 类型。这可以通过向转换添加 `output_field` 属性来实现：

```
from django.db.models import FloatField, Transform

class AbsoluteValue(Transform):
    lookup_name = 'abs'

    def as_sql(self, compiler, connection):
        lhs, params = compiler.compile(self.lhs)
        return "ABS(%s)" % lhs, params

    @property
    def output_field(self):
        return FloatField()
```

这确保了更进一步的查找，像 `abs__lte` 的行为和对 `FloatField` 表现的一样。

编写高效的 `abs__lt` 查找

当我们使用上面编写的 `abs` 查找的时候，在一些情况下，生成的SQL并不会高效使用索引。尤其是我们使用 `change__abs__lt=27` 的时候，这等价于 `change__gt=-27 AND change__lt=27` 。（对于 `lte` 的情况，我们可以使用SQL子句 `BETWEEN` ）。

所以我们想让 `Experiment.objects.filter(change__abs__lt=27)` 生成以下SQL:

```
SELECT .. WHERE "experiments"."change" < 27 AND "experiments"."c
hange" > -27
```

它的实现为：

```
from django.db.models import Lookup

class AbsoluteValueLessThan(Lookup):
    lookup_name = 'lt'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = compiler.compile(self.lhs.lhs)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params + lhs_params + rhs_para
ms
        return '%s < %s AND %s > -%s' % (lhs, rhs, lhs, rhs), pa
rams

AbsoluteValue.register_lookup(AbsoluteValueLessThan)
```

有一些值得注意的事情。首先， `AbsoluteValueLessThan` 并不调用 `process_lhs()` 。而是它跳过了由 `AbsoluteValue` 完成的 `lhs` ，并且使用原始的 `lhs` 。这就是说，我们想要得到 `27` 而不是 `ABS(27)` 。直接引用 `self.lhs.lhs` 是安全的，因为 `AbsoluteValueLessThan` 只能够通过 `AbsoluteValue` 查找来访问，这就是说 `lhs` 始终是 `AbsoluteValue` 的实例。

也要注意，就像两边都要在查询中使用多次一样，参数也需要多次包含 `lhs_params` 和 `rhs_params` 。

最终的实现直接在数据库中执行了反转 (27变为 -27) 。这样做的原因是如果 `self.rhs` 不是一个普通的整数值（比如是一个 `F()` 引用），我们在Python中不能执行这一转换。

> 注意
>
> 实际上，大多数带有__abs的查找都实现为这种范围查询，并且在大多数数据库后端中它更可能执行成这样，就像你可以利用索引一样。然而在PostgreSQL中，你可能想要向abs(change) 中添加索引，这会使查询更高效。

一个双向转换器的示例

我们之前讨论的， `AbsoluteValue` 的例子是一个只应用在查找左侧的转换。可能有一些情况，你想要把转换同时应用在左侧和右侧。比如，你想过滤一个基于左右侧相等比较操作的查询集，在执行一些SQL函数之后它们是大小写不敏感的。

让我们测试一下这一大小写不敏感的转换的简单示例。这个转换在实践中并不是十分有用，因为Django已经自带了一些自建的大小写不敏感的查找，但是它是一个很好的，数据库无关的双向转换示例。

我们定义使用SQL 函数 `UPPER()` 的 `UpperCase` 转换器，来在比较前转换这些值。我们定义了 `bilateral = True` 来表明转换同时作用在 `lhs` 和 `rhs` 上面：

```
from django.db.models import Transform

class UpperCase(Transform):
    lookup_name = 'upper'
    bilateral = True

    def as_sql(self, compiler, connection):
        lhs, params = compiler.compile(self.lhs)
        return "UPPER(%s)" % lhs, params
```

接下来，让我们注册它：

```
from django.db.models import CharField, TextField
CharField.register_lookup(UpperCase)
TextField.register_lookup(UpperCase)
```

现在，查询集 `Author.objects.filter(name__upper="doe")` 会生成像这样的大小写不敏感查询：

```
SELECT ... WHERE UPPER("author"."name") = UPPER('doe')
```

为现存查找编写自动的实现

有时不同的数据库供应商对于相同的操作需要不同的SQL。对于这个例子，我们会为MySQL重新编写一个自定义的， `NotEqual` 操作的实现。我们会使用 `!=` 而不是 `<>` 操作符。（注意实际上几乎所有数据库都支持这两个，包括所有Django支持的官方数据库）。

我们可以通过创建带有 `as_mysql` 方法的 `NotEqual` 的子类来修改特定后端上的行为。

```
class MySQLNotEqual(NotEqual):
    def as_mysql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s != %s' % (lhs, rhs), params

Field.register_lookup(MySQLNotEqual)
```

我们可以在 `Field` 中注册它。它取代了原始的 `NotEqual` 类，由于它具有相同的 `lookup_name` 。

当编译一个查询的时候，Django首先寻找 `as_%s % connection.vendor` 方法，然后回退到 `as_sql` 。内建后端的供应商名称是 sqlite，postgresql， oracle 和 mysql。

Django如何决定使用查找还是转换

有些情况下，你可能想要动态修改基于传递进来的名称， `Transform` 或者 `Lookup` 哪个会返回，而不是固定它。比如，你拥有可以储存搭配（ coordinate）或者任意一个维度（dimension）的字段，并且想让类似于 `.filter(coords__x7=4)` 的语法返回第七个搭配值为4的对象。为了这样做，你可以用一些东西覆写 `get_lookup` ，比如：

```
class CoordinatesField(Field):
    def get_lookup(self, lookup_name):
        if lookup_name.startswith('x'):
            try:
                dimension = int(lookup_name[1:])
            except ValueError:
                pass
            finally:
                return get_coordinate_lookup(dimension)
        return super(CoordinatesField, self).get_lookup(lookup_n
ame)
```

之后你应该合理定义 `get_coordinate_lookup` 。来返回一个 `Lookup` 的子类，它处理 `dimension` 的相关值。

有一个名称相似的方法叫做 `get_transform()` 。 `get_lookup()` 应该始终返回 `Lookup` 的子类，而 `get_transform()` 返回 `Transform` 的子类。记住 `Transform` 对象可以进一步过滤，而 `Lookup` 对象不可以，这非常重要。

过滤的时候，如果还剩下只有一个查找名称要处理，它会寻找 `Lookup` 。如果有多个名称，它会寻找 `Transform` 。在只有一个名称并且 Lookup找不到的情况下，会寻找 `Transform` ，之后寻找在 `Transform` 上面的 `exact` 查找。所有调用的语句都以一个 `Lookup` 结尾。解释一下：

- `.filter(myfield__mylookup)` 会调用 `myfield.get_lookup('mylookup')` 。
- `.filter(myfield__mytransform__mylookup)` 会调用 `myfield.get_transform('mytransform')` ，然后调用 `mytransform.get_lookup('mylookup')` 。
- `.filter(myfield__mytransform)` 会首先调用 `myfield.get_lookup('mytransform')` ，这样会失败，所以它会回退来调用 `myfield.get_transform('mytransform')` ，之后是 `mytransform.get_lookup('exact')` 。

> 译者：Django 文档协作翻译小组，原文：Custom lookups。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

查询表达式

查询表达式可以作为过滤，分组，注解或者是聚合的一个值或者是计算。这里（文档中）有很多内置表达式可以帮助你完成自己的查询。表达式可以组合，甚至是嵌套，来完成更加复杂的计算

支持的算术

Django支持在查询表达式使用加减乘除，求模，幂运算，Python常量，变量甚至是其它表达式。

New in Django 1.7:

增加了对指数运算符 `**` 的支持。

一些例子

Changed in Django 1.8:

一些例子依赖于Django1.8中的新功能

```
from django.db.models import F, Count
from django.db.models.functions import Length

# Find companies that have more employees than chairs.
Company.objects.filter(num_employees__gt=F('num_chairs'))

# Find companies that have at least twice as many employees
# as chairs. Both the querysets below are equivalent.
Company.objects.filter(num_employees__gt=F('num_chairs') * 2)
Company.objects.filter(
    num_employees__gt=F('num_chairs') + F('num_chairs'))

# How many chairs are needed for each company to seat all employ
ees?
>>> company = Company.objects.filter(
...    num_employees__gt=F('num_chairs')).annotate(
...    chairs_needed=F('num_employees') - F('num_chairs')).first
()
>>> company.num_employees
120
>>> company.num_chairs
50
>>> company.chairs_needed
70

# Annotate models with an aggregated value. Both forms
# below are equivalent.
Company.objects.annotate(num_products=Count('products'))
Company.objects.annotate(num_products=Count(F('products')))

# Aggregates can contain complex computations also
Company.objects.annotate(num_offerings=Count(F('products') + F('
services')))

# Expressions can also be used in order_by()
Company.objects.order_by(Length('name').asc())
Company.objects.order_by(Length('name').desc())
```

内置表达式

说明

这些表达式定义在 `django.db.models.expressions` 和 `django.db.models.aggregates` 中，但为了方便，通常可以直接从 `django.db.models` 导入。

`F()` 表达式

class `F`

一个 `F()` 对象代表了一个model的字段值或注释列。使用它就可以直接参考model的field和执行数据库操作而不用再把它们（model field）查询出来放到python内存中。

作为代替，Django使用 `F()` 对象生成一个SQL表达式，来描述数据库层级所需要的操作

这些通过一个例子可以很容易的理解。往常，我们会这样做：

```
# Tintin filed a news story!
reporter = Reporters.objects.get(name='Tintin')
reporter.stories_filed += 1
reporter.save()
```

这里呢，我们把 `reporter.stories_filed` 的值从数据库取出放到内存中并用我们熟悉的python运算符操作它，最后再把它保存到数据库。然而，我们还可以这样做：

```
from django.db.models import F

reporter = Reporters.objects.get(name='Tintin')
reporter.stories_filed = F('stories_filed') + 1
reporter.save()
```

虽然 `reporter.stories_filed = F('stories_filed') + 1` 看起来像一个正常的Python分配值赋给一个实例属性，事实上这是一个描述数据库操作的SQL概念

当Django遇到 `F()` 实例，它覆盖了标准的Python运算符创建一个封装的SQL表达式。在这个例子中， `reporter.stories_filed` 就代表了一个指示数据库对该字段进行增量的命令。

无论 `reporter.stories_filed` 的值是或曾是什么，Python一无所知--这完全是由数据库去处理的。所有的Python，通过Django的 `F()` 类，只是去创建SQL语法参考字段和描述操作。

注

为了获得用这种方法保存的新值，此对象应重新加载：

```
reporter = Reporters.objects.get(pk=reporter.pk)
```

.和上面单独实例的操作一样， `F()` 配合 `update()`可以应用于对象实例的 QuerySets `这减少了我们上面使用的两个查询 -` get() 和[`save()`](instances.html#django.db.models.Model.save "django.db.models.Model.save") - 只有一个：

```
reporter = Reporters.objects.filter(name='Tintin')
reporter.update(stories_filed=F('stories_filed') + 1)
```

我们可以使用 `update()` 方法批量地增加多个对象的字段值。这比先从数据库查询后，通过循环一个个增加，并一个个保存要快的很多。

```
Reporter.objects.all().update(stories_filed=F('stories_filed') +
 1)
```

`F()` 表达式的效率上的优点主要体现在

- 直接通过数据库操作而不是Python
- 减少数据库查询次数

使用 `F()` 避免竞态条件

使用 `F()` 的另一个好处是通过数据库而不是Python来更新字段值以避免竞态条件.

如果两个Python线程执行上面第一个例子中的代码，一个线程可能在另一个线程刚从数据库中获取完字段值后获取、增加、保存该字段值。第二个线程保存的值将会基于原始字段值；第一个线程的工作将会丢失。

如果让数据库对更新字段负责，这个过程将变得更稳健：它将只会在 `save()` 或 `update()` 执行时根据数据库中该字段值来更新字段，而不是基于实例之前获取的字段值。

在过滤器中使用 `F()`

`F()` 在 `查询集` 过滤器中也十分有用，它使得使用条件通过字段值而不是Python值过滤一组对象变得可能。

这在 在查询中使用*F()*表达式 中被记录。

使用 `F()` 和注释

`F()` 可用于通过将不同字段与算术相结合来在模型上创建动态字段：

```
company = Company.objects.annotate(
    chairs_needed=F('num_employees') - F('num_chairs'))
```

如果你组合的字段是不同类型，你需要告诉Django将返回什么类型的字段。由于 `F()` 不直接支持 `output_field` ，您需要使用 `ExpressionWrapper`

```
from django.db.models import DateTimeField, ExpressionWrapper, F

Ticket.objects.annotate(
    expires=ExpressionWrapper(
        F('active_at') + F('duration'), output_field=DateTimeFie
ld()))
```

`Func()` 表达式

New in Django 1.8.

`Func()` 表达式是所有表达式的基础类型，包括数据库函数如 `COALESCE` 和 `LOWER` , 或者 `SUM` 聚合.用下面方式可以直接使用:

```
from django.db.models import Func, F

queryset.annotate(field_lower=Func(F('field'), function='LOWER')
)
```

或者它们可以用于构建数据库函数库：

```
class Lower(Func):
    function = 'LOWER'

queryset.annotate(field_lower=Lower(F('field')))
```

但是这两种情况都将导致查询集，其中每个模型用从以下SQL大致生成的额外属性 `field_lower` 注释：

```
SELECT
    ...
    LOWER("app_label"."field") as "field_lower"
```

有关内置数据库函数的列表，请参见数据库函数。

`Func` API如下：

class `Func` (**expressions*, ***extra*)

`function`

描述将生成的函数的类属性。具体来说， `函数` 将被插入为 `模板` 中的 `函数` 占位符。默认为 `无` 。

`template`

类属性，作为格式字符串，描述为此函数生成的SQL。默认为 `'%（function）s（%（expressions）s）'` 。

`arg_joiner`

类属性，表示用于连接 `表达式` 列表的字符。默认为 `'， '` 。

`*expressions` 参数是函数将要应用与表达式的位置参数列表。表达式将转换为字符串，与 `arg_joiner` 连接在一起，然后作为 `expressions` 占位符插入 `template` 。

位置参数可以是表达式或Python值。字符串假定为列引用，并且将包装在 `F()` 表达式中，而其他值将包裹在 `Value()` 表达式中。

`**extra` kwargs是可以插入到 `template` 属性中的 `key=value` 对。请注意，关键字 `function` 和 `template` 可用于分别替换 `function` 和 `template` 属性，而无需定义自己的类。 `output_field` 可用于定义预期的返回类型。

`Aggregate()` 表达式

聚合表达式是*Func() expression*的一种特殊情况，它通知查询： `GROUP BY` 子句是必须的。所有*aggregate functions*，如 `Sum()` 和 `Count()` ，继承自 `Aggregate()` 。

由于 `Aggregate` 是表达式和换行表达式，因此您可以表示一些复杂的计算：

```
from django.db.models import Count

Company.objects.annotate(
    managers_required=(Count('num_employees') / 4) + Count('num_
managers'))
```

`Aggregate` API如下：

class `Aggregate` (*expression*, *output_field=None*, ***extra*)

`template`

类属性，作为格式字符串，描述为此聚合生成的SQL。默认为 `'%（函数）s（ %（表达式）s ）'` 。

`function`

描述将生成的聚合函数的类属性。具体来说， `function` 将被插入为 `template` 中的 `function` 占位符。默认为 `None` 。

`expression` 参数可以是模型上的字段的名称，也可以是另一个表达式。它将转换为字符串，并用作 `template` 中的 `expressions` 占位符。

`output_field` 参数需要一个模型字段实例，如 `IntegerField()` 或 `BooleanField()` ，Django将在检索后数据库。通常在实例化模型字段时不需要任何参数，因为与数据验证相关的任何参数（ `max_length` ， `max_digits` 等）不会对表达式的输出值强制执行。

注意，只有当Django无法确定结果应该是什么字段类型时，才需要 `output_field` 。混合字段类型的复杂表达式应定义所需的 `output_field` 。例如，将 `IntegerField()` 和 `FloatField()` 添加在一起应该可以有 `output_field=FloatField()`

Changed in Django 1.8:

`output_field` 是一个新参数。

`**extra` kwargs是可以插入到 `template` 属性中的 `key=value` 对。

New in Django 1.8:

聚合函数现在可以在单个函数中使用算术和参考多个模型字段。

创建自己的聚合函数

创建自己的聚合是非常容易的。至少，您需要定义 `function` ，但也可以完全自定义生成的SQL。这里有一个简单的例子：

```
from django.db.models import Aggregate

class Count(Aggregate):
    # supports COUNT(distinct field)
    function = 'COUNT'
    template = '%(function)s(%(distinct)s%(expressions)s)'

    def __init__(self, expression, distinct=False, **extra):
        super(Count, self).__init__(
            expression,
            distinct='DISTINCT ' if distinct else '',
            output_field=IntegerField(),
            **extra)
```

`Value()` 表达式

class `Value` (*value*, *output_field=None*)

`Value()` 对象表示表达式的最小可能组件：简单值。当您需要在表达式中表示整数，布尔或字符串的值时，可以在 `Value()` 中包装该值。

您很少需要直接使用 `Value()` 。当您编写表达式 `F('field') + 1` 时，Django隐式包装 `1` 在 `Value()` 中，允许在更复杂的表达式中使用简单的值。

`value` 参数描述要包括在表达式中的值，例如 `1` ， `True` 或 `None` 。Django知道如何将这些Python值转换为相应的数据库类型。

`output_field` 参数应为模型字段实例，如 `IntegerField()` 或 `BooleanField()` ，Django将在检索后从数据库。通常在实例化模型字段时不需要任何参数，因为与数据验证相关的任何参数（ `max_length` ， `max_digits` 等）不会对表达式的输出值强制执行。

`ExpressionWrapper()` 表达式

class `ExpressionWrapper` (*expression*, *output_field*)

New in Django 1.8.

`ExpressionWrapper` 简单地包围另一个表达式，并提供对其他表达式可能不可用的属性（例如 `output_field` ）的访问。当对*Using F() with annotations*中描述的不同类型的 `F()` 表达式使用算术时，必须使用 `ExpressionWrapper` 。

条件表达式

New in Django 1.8.

条件表达式允许您使用 `if` ... `elif` ... `else` 查询中的逻辑。Django本地支持SQL `CASE` 表达式。有关更多详细信息，请参阅*Conditional Expressions*。

技术信息

下面您将找到对图书馆作者有用的技术实施细节。下面的技术API和示例将有助于创建可扩展Django提供的内置功能的通用查询表达式。

表达式API

查询表达式实现了*query expression API*，但也暴露了下面列出的一些额外的方法和属性。所有查询表达式必须从 `Expression()` 或相关子类继承。

当查询表达式包装另一个表达式时，它负责调用包装表达式上的相应方法。

class `Expression`

`contains_aggregate`

告诉Django此表达式包含聚合，并且需要将 `GROUP BY` 子句添加到查询中。

`resolve_expression` (*query=None*, *allow_joins=True*, *reuse=None*, *summarize=False*)

提供在将表达式添加到查询之前对表达式执行任何预处理或验证的机会。还必须在任何嵌套表达式上调用 `resolve_expression()` 。应该返回具有任何必要变换的 `self` 的 `copy()` 。

`query` 是后端查询实现。

`allow_joins` 是一个布尔值，允许或拒绝在查询中使用联接。

`reuse` 是用于多连接场景的一组可重用连接。

`summarize` 是一个布尔值，当 `True` 时，表示正在计算的查询是终端聚合查询。

`get_source_expressions` ()

返回内部表达式的有序列表。例如：

```
&gt;&gt;&gt; Sum(F('foo')).get_source_expressions()
[F('foo')]
```

`set_source_expressions` (*expressions*)

获取表达式列表并存储它们，以便 `get_source_expressions()` 可以返回它们。

`relabeled_clone` (*change_map*)

返回 `self` 的克隆（副本），并重新标记任何列别名。创建子查询时，将重命名列别名。 `relabeled_clone()` 也应该在任何嵌套表达式上调用并分配给克隆。

`change_map` 是将旧别名映射到新别名的字典。

例：

```
def relabeled_clone(self, change_map):
    clone = copy.copy(self)
    clone.expression = self.expression.relabeled_clone(change_ma
p)
    return clone
```

`convert_value` (*self*, *value*, *expression*, *connection*, *context*)

允许表达式将 `value` 强制为更适当类型的钩子。

`refs_aggregate` (*existing_aggregates*)

Returns a tuple containing the `(aggregate, lookup_path)` of the first aggregate that this expression (or any nested expression) references, or `(False, ())` if no aggregate is referenced. 例如：

```
queryset.filter(num_chairs__gt=F('sum__employees'))
```

`F()` 表达式此处引用前一个 `Sum()` 计算，这意味着此过滤器表达式应添加到 `HAVING` `WHERE` 子句。

在大多数情况下，对任何嵌套表达式返回 `refs_aggregate` 的结果应该是合适的，因为必要的内置表达式将返回正确的值。

`get_group_by_cols` ()

负责返回此表达式的列引用列表。 `get_group_by_cols()` 应在任何嵌套表达式上调用。 `F()` 对象，特别是保存对列的引用。

`asc` ()

返回准备好以升序排序的表达式。

`desc` ()

返回准备好以降序排序的表达式。

`reverse_ordering` ()

通过在 `order_by` 调用中反转排序顺序所需的任何修改，返回 `self` 。例如，执行 `NULLS LAST` 的表达式将其值更改为 `NULLS FIRST` 。仅对实现类似 `OrderBy` 的排序顺序的表达式需要修改。当在查询集上调用 `reverse()` 时，调用此方法。

编写自己的查询表达式

您可以编写自己的查询表达式类，这些类使用其他查询表达式，并可以与其集成。让我们通过编写一个 `COALESCE` SQL函数的实现，而不使用内置的*Func() expressions*来演示一个例子。

`COALESCE` SQL函数定义为获取列或值的列表。它将返回不是 `NULL` 的第一列或值。

我们将首先定义用于生成SQL的模板，然后使用 `__init__()` 方法来设置一些属性：

```
import copy
from django.db.models import Expression

class Coalesce(Expression):
    template = 'COALESCE( %(expressions)s )'

    def __init__(self, expressions, output_field, **extra):
      super(Coalesce, self).__init__(output_field=output_field)
      if len(expressions) < 2:
          raise ValueError('expressions must have at least 2 ele
ments')
      for expression in expressions:
          if not hasattr(expression, 'resolve_expression'):
              raise TypeError('%r is not an Expression' % expres
sion)
      self.expressions = expressions
      self.extra = extra
```

我们对参数进行一些基本验证，包括至少需要2列或值，并确保它们是表达式。我们在这里需要 `output_field` ，以便Django知道要将最终结果分配给什么样的模型字段。

现在我们实现预处理和验证。由于我们现在没有任何自己的验证，我们只是委托给嵌套表达式：

```
def resolve_expression(self, query=None, allow_joins=True, reuse
=None, summarize=False):
    c = self.copy()
    c.is_summary = summarize
    for pos, expression in enumerate(self.expressions):
        c.expressions[pos] = expression.resolve_expression(query
, allow_joins, reuse, summarize)
    return c
```

接下来，我们编写负责生成SQL的方法：

```
def as_sql(self, compiler, connection):
    sql_expressions, sql_params = [], []
    for expression in self.expressions:
        sql, params = compiler.compile(expression)
        sql_expressions.append(sql)
        sql_params.extend(params)
    self.extra['expressions'] = ','.join(sql_expressions)
    return self.template % self.extra, sql_params

def as_oracle(self, compiler, connection):
    """
 Example of vendor specific handling (Oracle in this case).
 Let's make the function name lowercase.
 """
    self.template = 'coalesce( %(expressions)s )'
    return self.as_sql(compiler, connection)
```

我们使用 `compiler.compile()` 方法为每个 `expressions` 生成SQL，并用逗号连接结果。然后使用我们的数据填充模板，并返回SQL和参数。

我们还定义了一个特定于Oracle后端的自定义实现。如果Oracle后端正在使用，则将调用 `as_oracle()` 函数，而不是 `as_sql()` 。

最后，我们实现允许我们的查询表达式与其他查询表达式一起播放的其他方法：

```
def get_source_expressions(self):
    return self.expressions

def set_source_expressions(self, expressions):
    self.expressions = expressions
```

让我们看看它是如何工作的：

```
>>> from django.db.models import F, Value, CharField
>>> qs = Company.objects.annotate(
...    tagline=Coalesce([
...        F('motto'),
...        F('ticker_name'),
...        F('description'),
...        Value('No Tagline')
...        ], output_field=CharField()))
>>> for c in qs:
...     print("%s: %s" % (c.name, c.tagline))
...
Google: Do No Evil
Apple: AAPL
Yahoo: Internet Company
Django Software Foundation: No Tagline
```

条件表达式

```
New in Django 1.8.
```

条件表达式允许你在过滤器、注解、聚合和更新操作中使用 `if ... elif ... else` 的逻辑。条件件表达式为表中的每一行计算一系列的条件，并且返回匹配到的结果表达式。条件表达式也可以像其它 表达式一样混合和嵌套。

条件表达式类

我们会在后面的例子中使用下面的模型：

```
from django.db import models

class Client(models.Model):
    REGULAR = 'R'
    GOLD = 'G'
    PLATINUM = 'P'
    ACCOUNT_TYPE_CHOICES = (
        (REGULAR, 'Regular'),
        (GOLD, 'Gold'),
        (PLATINUM, 'Platinum'),
    )
    name = models.CharField(max_length=50)
    registered_on = models.DateField()
    account_type = models.CharField(
        max_length=1,
        choices=ACCOUNT_TYPE_CHOICES,
        default=REGULAR,
    )
```

When

`class When(condition=None, then=None, **lookups)[source]`

`When()` 对象用于封装条件和它的结果，为了在条件表达式中使用。使用 `When()` 对象和使用 `filter()` 方法类似。条件可以使用字段查找 或者 `Q` 来指定。结果通过使用 `then` 关键字来提供。

一些例子：

```
>>> from django.db.models import When, F, Q
>>> # String arguments refer to fields; the following two exampl
es are equivalent:
>>> When(account_type=Client.GOLD, then='name')
>>> When(account_type=Client.GOLD, then=F('name'))
>>> # You can use field lookups in the condition
>>> from datetime import date
>>> When(registered_on__gt=date(2014, 1, 1),
...      registered_on__lt=date(2015, 1, 1),
...      then='account_type')
>>> # Complex conditions can be created using Q objects
>>> When(Q(name__startswith="John") | Q(name__startswith="Paul")
,
...      then='name')
```

要注意这些值中的每一个都可以是表达式。

> 注意
>
> 由于then 关键字参数为 When()的结果而保留，如果Model有名称为 then的字段，会有潜在的冲突。这可以用以下两种办法解决：

```
>>> from django.db.models import Value
>>> When(then__exact=0, then=1)
>>> When(Q(then=0), then=1)
```

Case

`class Case(*cases, **extra)[source]`

`Case()` 表达式就像是Python中的 `if ... elif ... else` 语句。每个提供的 `When()` 中的 `condition` 按照顺序计算，直到得到一个真值。返回匹配 `When()` 对象的 `result` 表达式。

一个简单的例子：

```
>>>
>>> from datetime import date, timedelta
>>> from django.db.models import CharField, Case, Value, When
>>> Client.objects.create(
...     name='Jane Doe',
...     account_type=Client.REGULAR,
...     registered_on=date.today() - timedelta(days=36))
>>> Client.objects.create(
...     name='James Smith',
...     account_type=Client.GOLD,
...     registered_on=date.today() - timedelta(days=5))
>>> Client.objects.create(
...     name='Jack Black',
...     account_type=Client.PLATINUM,
...     registered_on=date.today() - timedelta(days=10 * 365))
>>> # Get the discount for each Client based on the account type
>>> Client.objects.annotate(
...     discount=Case(
...         When(account_type=Client.GOLD, then=Value('5%')),
...         When(account_type=Client.PLATINUM, then=Value('10%')
),
...         default=Value('0%'),
...         output_field=CharField(),
...     ),
... ).values_list('name', 'discount')
[('Jane Doe', '0%'), ('James Smith', '5%'), ('Jack Black', '10%'
)]
```

`Case()` 接受任意数量的 `When()` 对象作为独立的参数。其它选项使用关键字参数提供。如果没有条件为 `TRUE` ，表达式会返回提供的 `default` 关键字参数。如果没有提供 `default` 参数，会使用 `Value(None)` 。

如果我们想要修改之前的查询，来获取基于 `Client` 跟着我们多长时间的折扣，我们应该这样使用查找：

```
>>> a_month_ago = date.today() - timedelta(days=30)
>>> a_year_ago = date.today() - timedelta(days=365)
>>> # Get the discount for each Client based on the registration
 date
>>> Client.objects.annotate(
...     discount=Case(
...         When(registered_on__lte=a_year_ago, then=Value('10%'
)),
...         When(registered_on__lte=a_month_ago, then=Value('5%'
)),
...         default=Value('0%'),
...         output_field=CharField(),
...     )
... ).values_list('name', 'discount')
[('Jane Doe', '5%'), ('James Smith', '0%'), ('Jack Black', '10%'
)]
```

> 注意
>
> 记住条件按照顺序来计算，所以上面的例子中，即使第二个条件匹配到了 Jane Doe 和 Jack Black，我们也得到了正确的结果。这就像Python中的if ... elif ... else语句一样。

高级查询

条件表达式可以用于注解、聚合、查找和更新。它们也可以和其它表达式混合和嵌套。这可以让你构造更强大的条件查询。

条件更新

假设我们想要为客户端修改 `account_type` 来匹配它们的注册日期。我们可以使用条件表达式和 `update()` 放啊来实现：

```
>>> a_month_ago = date.today() - timedelta(days=30)
>>> a_year_ago = date.today() - timedelta(days=365)
>>> # Update the account_type for each Client from the registrat
ion date
>>> Client.objects.update(
...     account_type=Case(
...         When(registered_on__lte=a_year_ago,
...              then=Value(Client.PLATINUM)),
...         When(registered_on__lte=a_month_ago,
...              then=Value(Client.GOLD)),
...         default=Value(Client.REGULAR)
...     ),
... )
>>> Client.objects.values_list('name', 'account_type')
[('Jane Doe', 'G'), ('James Smith', 'R'), ('Jack Black', 'P')]
```

条件聚合

如果我们想要弄清楚每个 `account_type` 有多少客户端，要怎么做呢？我们可以在聚合函数中嵌套条件表达式来实现：

```
>>> # Create some more Clients first so we can have something to
 count
>>> Client.objects.create(
...     name='Jean Grey',
...     account_type=Client.REGULAR,
...     registered_on=date.today())
>>> Client.objects.create(
...     name='James Bond',
...     account_type=Client.PLATINUM,
...     registered_on=date.today())
>>> Client.objects.create(
...     name='Jane Porter',
...     account_type=Client.PLATINUM,
...     registered_on=date.today())
>>> # Get counts for each value of account_type
>>> from django.db.models import IntegerField, Sum
>>> Client.objects.aggregate(
...     regular=Sum(
...         Case(When(account_type=Client.REGULAR, then=1),
...              output_field=IntegerField())
...     ),
...     gold=Sum(
...         Case(When(account_type=Client.GOLD, then=1),
...              output_field=IntegerField())
...     ),
...     platinum=Sum(
...         Case(When(account_type=Client.PLATINUM, then=1),
...              output_field=IntegerField())
...     )
... )
{'regular': 2, 'gold': 1, 'platinum': 3}
```

> 译者：Django 文档协作翻译小组，原文：Conditional Expressions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

数据库函数

```
New in Django 1.8.
```

下面记述的类为用户提供了一些方法，来在Django中使用底层数据库提供的函数用于注解、聚合或者过滤器等操作。函数也是表达式，所以可以像聚合函数一样混合使用它们。

我们会在每个函数的实例中使用下面的模型：

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    age = models.PositiveIntegerField(null=True, blank=True)
    alias = models.CharField(max_length=50, null=True, blank=Tru
e)
    goes_by = models.CharField(max_length=50, null=True, blank=T
rue)
```

我们并不推荐在CharField上允许null=True，以后那位这会允许字段有两个“空值”，但是对于下面的Coalesce示例来说它很重要。

Coalesce

`class Coalesce(*expressions, **extra)[source]`

接受一个含有至少两个字段名称或表达式的列表，返回第一个非空的值（注意空字符串不被认为是一个空值）。每个参与都必须是相似的类型，所以掺杂了文本和数字的列表会导致数据库错误。

使用范例：

```
>>> # Get a screen name from least to most public
>>> from django.db.models import Sum, Value as V
>>> from django.db.models.functions import Coalesce
>>> Author.objects.create(name='Margaret Smith', goes_by='Maggie
')
>>> author = Author.objects.annotate(
...    screen_name=Coalesce('alias', 'goes_by', 'name')).get()
>>> print(author.screen_name)
Maggie

>>> # Prevent an aggregate Sum() from returning None
>>> aggregated = Author.objects.aggregate(
...    combined_age=Coalesce(Sum('age'), V(0)),
...    combined_age_default=Sum('age'))
>>> print(aggregated['combined_age'])
0
>>> print(aggregated['combined_age_default'])
None
```

Concat

`class Concat(*expressions, **extra)[source]`

接受一个含有至少两个文本字段的或表达式的列表，返回连接后的文本。每个参数都必须是文本或者字符类型。如果你想把一个 `TextField()` 和一个 `CharField()` 连接， 一定要告诉Django `output_field` 应该为 `TextField()` 类型。在下面连接 `Value` 的例子中，这也是必需的。

这个函数不会返回 `null` 。在后端中，如果一个 `null` 参数导致了整个表达式都是 `null` ，Django会确保把每个 `null` 的部分转换成一个空字符串。

使用范例：

```
>>> # Get the display name as "name (goes_by)"
>>> from django.db.models import CharField, Value as V
>>> from django.db.models.functions import Concat
>>> Author.objects.create(name='Margaret Smith', goes_by='Maggie
')
>>> author = Author.objects.annotate(
...    screen_name=Concat('name', V(' ('), 'goes_by', V(')'),
...    output_field=CharField())).get()
>>> print(author.screen_name)
Margaret Smith (Maggie)
```

Length

`class Length(expression, **extra)[source]`

接受一个文本字段或表达式，返回值的字符个数。如果表达式是 `null` ，长度也会是 `null` 。

使用范例：

```
>>> # Get the length of the name and goes_by fields
>>> from django.db.models.functions import Length
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(
...    name_length=Length('name'),
...    goes_by_length=Length('goes_by')).get()
>>> print(author.name_length, author.goes_by_length)
(14, None)
```

Lower

`class Lower(expression, **extra)[source]`

接受一个文本字符串或表达式，返回它的小写表示形式。

使用范例：

```
>>> from django.db.models.functions import Lower
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(name_lower=Lower('name')).g
et()
>>> print(author.name_lower)
margaret smith
```

Substr

`class Substr(expression, pos, length=None, **extra)[source]`

返回这个字段或者表达式的，以 `pos` 位置开始，长度为 `length` 的子字符串。位置从下标为1开始，所以必须大于0。如果 `length` 是 `None` ，会返回剩余的字符串。

使用范例：

```
>>> # Set the alias to the first 5 characters of the name as low
ercase
>>> from django.db.models.functions import Substr, Lower
>>> Author.objects.create(name='Margaret Smith')
>>> Author.objects.update(alias=Lower(Substr('name', 1, 5)))
1
>>> print(Author.objects.get(name='Margaret Smith').alias)
marga
```

Upper

`class Upper(expression, **extra)[source]`

接受一个文本字符串或表达式，返回它的大写表示形式。

使用范例：

```
>>> from django.db.models.functions import Upper
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(name_upper=Upper('name')).g
et()
>>> print(author.name_upper)
MARGARET SMITH
```

> 译者：Django 文档协作翻译小组，原文：Database Functions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

其它

数据库

Django试图尽可能多的支持所有数据库后端的特性。然而，并不是所有数据库都一样，所以我们必须在支持哪些特性和做出哪些安全的假定上做出设计决策。

本文描述了一些Django使用数据库的有关特性。当然，它并不想成为各服务器指定的文档或者参考手册的替代品。

综合说明

持续连接特性

持续连接的特性避免了每一次重新建立与数据库的连接的请求中所增加的压力。这些连接通过 `CONN_MAX_AGE` 参数(控制一个连接的最长存活时间)来控制。它可以被单独的设置在每一个数据库中。

参数的默认值为 `0` ，对每次请求结束时终止数据库连接的历史行为提供保护。当启用持续连接时，设置 `CONN_MAX_AGE` 的值(正数，单位为秒)为每个连接的最大存活时间。对于无限制的连接，请设置为 `None` 。

连接管理

Django在它第一次建立数据库查询的时候打开与数据库的连接。它保持着连接的通畅使得在随后的请求中可以再利用。一旦这个连接超过参数 `CONN_MAX_AGE` 的最长存活时间的限制或者不能更长久的存活时，Django就会终止连接。

详细地说，当Django需要的时或者当前不存在数据库连接时，它会自动的与数据库之间建立连接————要么是因为这是第一次连接，要么是因为先前连接被关闭。

在每个请求开始时后，如果连接到了设置的最长存活时间时，Django就会关闭它。如果你的数据库在之后一段时间终止了闲置的连接，你应该设置 `CONN_MAX_AGE` 为一个更低的值，以至于Django不会尝试去连接数据库服务器上已经终止的连接。（这个问题可能仅仅影响流量比较低的站点。）

在每个请求结束，如果它已经到达了最长存活时间或者它处在一个不可恢复的错误状态时，Django就会关闭连接。在处理请求中如果有任何数据库错误产生，Django就会检查是否连接仍然工作，不工作就关闭它。因此，数据库错误最多影响一个请求。如果连接不可用，下一个 请求会打开一个新的连接。

注意事项

因为每个线程都保持着自己的连接，你的数据库至少必须在你的工作线程中支持尽可能多的并发线程。

有的时候数据库并不会在你的视图中有太多的访问，例如，因为数据库是一个外部系统，或者因为缓存的作用。在这种情况下，你应该设置 `CONN_MAX_AGE` 为一个低的值或者为 `0` ，因为这对保持一个连接并没有意义，而且连接不太可能被重复调用。这会有助于保持数据库的并发性连接数为一个小的值。

开发服务器对每个需要处理的请求创建一个新的线程，用来否定持续连接的影响。不要在开发过程中启用他们。

当Django建立与数据库的连接时，它会设置相应的参数，这取决于后台的使用情况。如果你启用了持续性连接，那么程序将不再重复每个请求。如果你修改了参数，例如连接的隔离级别或者是时区，你也应该恢复Django原有的默认值在每个请求结束时，强制使用一个合适的值在每个请求开始时，或者禁用持续性连接。

编码

Django假定所有的数据库使用UTF-8编码。使用其他的编码有可能会导致不可预知的行为发生，例如在Django中你的数据库里有效的数据可能会出现“value too long”的错误。在下面的信息中你可以查看数据库的具体说明来正确的设置你的数据库。

PostgreSQL 说明

Django支持PostgreSQL 9.0和更高版本。它需要使用psycopg2 2.4.5或更高版本(或者 2.5+ 如果你想使用 `django.contrib.postgres`).

如果你使用的是Windows操作系统，请查看一下我们非官方psycopg2的compiled Windows version

PostgreSQL 连接设置

参考 `HOST` 来了解详细信息。

Optimizing PostgreSQL’s configuration

Django规定它的数据库连接需要下列参数：

- `client_encoding` : `'UTF8'` ,
- `default_transaction_isolation` : `'read committed'` 为默认值，或者是连接选项中设置的值（参见下文），
- `timezone` : `'UTC'` when `USE_TZ` is `True` , value of `TIME_ZONE` otherwise.

如果这些参数已经有了正确的值，Django将不会为每一个新的连接设置它们，这略微提高了性能。你可以直接在 `postgresql.conf` 中配置它们或者更方便的使用ALTER ROLE来为每一个database user(...设置?).

没有这个优化，Django也会工作得很好，但每一个新的连接会做一些额外的查询来设置这些参数。

隔离级别

像 PostgreSQL 本身，Django默认 `READ COMMITTED` 隔离级别isolation level. 如果你需要一个更高的隔离级别，例如 `REPEATABLE READ` 或者 `SERIALIZABLE`，在 `数据库databases` `OPTIONS` 设置部分的数据库配置：

```
import psycopg2.extensions

DATABASES = {
    # ...
    'OPTIONS': {
        'isolation_level': psycopg2.extensions.ISOLATION_LEVEL_S
ERIALIZABLE,
    },
}
```

说明

在较高的隔离级别中，你的application应该做好准备去处理序列化失败中引发的异常(？不确定)。这个选项被设计为高级的用法。

`varchar` 和 `text` 列的索引

当你在你的模块字段中指定了 `db_index=True`，Django通常会输出一个单一的 `CREATE INDEX` 语句。但是，如果数据库类型对应的字段是 `varchar` 或者 `text` (e.g., used by `CharField`, `FileField`, and `TextField`), 针对该列，Django会使用适当的PostgreSQL operator class 创建一个additional index.使用 `LIKE` 操作在SQL中,额外的索引是执行正确的查找所必需的，即用 `contains` and `startswith` 查找类型。

MySQL 说明

版本支持

Django 支持MySQL 5.5 和更高版本。

Django的 `inspectdb` 功能使用了 `information_schema` database,它在所有的database schemas包含了详细的数据。

Django希望数据库支持Unicode (UTF-8编码),并且代理它去执transactions and referential integrity的任务。当你使用MyISAM储存引擎时，一个你需要注意到的事情是在MYSQL，后两个实际上是不去执行的，请参阅下一节。

存储引擎

MySQL有数种存储引擎. 你可以改变默认的存储引擎在不同的配置中。

直到MySQL 5.5.4版本, 默认存数引擎是MyISAM [1]. MyISAM数据的主要缺点是，它不支 transactions或者执行foreign-key约束。从好的方面来看，直到MySQL 5.6.4，它是唯一一个支持全文索引和搜索的引擎。

自从MySQL 5.5.5,默认的储存引擎变为了 InnoDB. 这个引擎是全事务和支持foreign key引用的。这可能是目前最好的选择。但是，请注意由于它不能够记取 `AUTO_INCREMENT` 的值，而不是重新创建像 “max(id)+1”这样，导致InnoDB的autoincrement counter在MySQL中丢失了。这可能导致一个无意重用的 `AutoField`

如果将现有项目升级到MySQL5.5.5，随后添加一些表，确保您的表可以使用相同的存储引擎（如MyISAM数据与InnoDB的）。特别注意，如果你的表中存在 `ForeignKey` ，比较他们在不同的存储引擎中,你可能会看到如下的错误当运行 `migrate` 的时候:

```
_mysql_exceptions.OperationalError: (
    1005, "Can't create table '\\db_name\\.#sql-4a8_ab' (errno:
150)"
)
```

[1]	Unless this was changed by the packager of your MySQL package. We’ve had reports that the Windows Community Server installer sets up InnoDB as the default storage engine, for example.

MySQL DB API 驱动

Python的Database API 被描述为**PEP 249**. MySQL拥有三种很棒的能够实现API的驱动：

- MySQLdb是一个由Andy Dustman开发，已经发展并支持十多年的一个本地驱动。
- mysqlclient是 `MySQLdb` 的一个分支，它与python3有着特别好的契合并且可以作为MySQLdb的直接替代。在书写这篇的时候，这是在Django使用MySQL的推荐的选择。
- MySQL Connector/Python是一个来自Oracle的纯python驱动，它不需要MySQL client库或在标准库之外的任何Python模块。

所有这些驱动都是线程安全的，并提供连接池。 `MySQLdb` 是当前唯一一个不支持python3的。

除了一个DB API驱动，Django需要一个适配器来通过它的ORM访问数据库驱动。Django对 MySQLdb/mysqlclient提供了适配器直到MySQL Connector/Python包含了its own。

MySQLdb

Django需要MySQLdb version 1.2.1p2或更高版本。

在写文档的时候，最新版本MySQLdb、(1.2.5)还没有支持python3为了在Python 3使用MySQLdb, 你需要安装 `mysqlclient` 来替代它 。

说明

关于将date strings转化为 datetime objects，MySQLdb有已经存在的问题。具体来说，值为 `0000-00-00` 的date strings对于MySQL是有效的，但是会被MySQLdb转换成 `None` 。

这意味着当你在列中使用值可能为 `0000-00-00` 的 `loaddata` 和 `dumpdata` 时，你需要注意，因为它们会被转换为 `None` 。

mysqlclient

Django需要mysqlclient 1.3.3或更高版本。需要注意，Python 3.2 不支持。除了Python 3.3+ 支持, mysqlclient在其他版本表现的更相似于 MySQLDB。

MySQL Connector/Python

MySQL Connector/Python可从download page下载。 Django适配器在1.1.X或更高版本可用。它可能不支持最新的Django的版本。

时区规定

如果您计划使用Django的*timezone support*，使用 *mysql_tzinfo_to_sql*加载时区表到*MySQL*数据库。这需要在你的MySQL服务器部署一次就好，而不是在每个数据库上。

创建数据库

你可以使用命令行工具运行这条SQL语句来create your database:

```
CREATE DATABASE <dbname> CHARACTER SET utf8;
```

这可以确保所有表和列默认使用UTF-8。

排序设置

一列的排序规则设置控制了数据排序的顺序，以及字符串的比较。它可以被设置在数据库的水平，也可以在每一个表和每一列这个是在MySQL文档中详细记载的。在所有情况下，你可以直接操作数据库中的表来设置排序。Django不提供在模型定义

上设置它的方式。

默认情况下，在一个UTF8编码的数据库中，MySQL将会使用 `utf8_general_ci` 排序规则。这会导致所有的字符串比较是否相等的结果是以不区分大小写 的方式完成的。就像这样， `"Fred"` and `"freD"` 被认为是相等的在数据库级别上。如果你在字段里有一个特殊的约束，如果试图将 `"aa"` 和 `"AA"` 插入到同一列会是非法的，因为他们在默认的排序(and, hence, non-unique) 中比较相等。

在许多情况下，这种默认情况将不会成为一个问题。然而,如果你真的想要区分大小写比较在一个特定的列或表,你要把这个列或表应用 `utf8_bin` 排序。主要的在这种情况下需要注意的是,如果您正在使用MySQLdb 1.2.2 Django的数据库后端会返回bytestrings(而非unicode字符串)对于从数据库中获得的任何字符字段。Django平常的做法总是返回unicode字符串,与之相比这是一个显著的变化。这取决于你，开发者们，事实中你将会得到bytestrings的返回如果你配置你的表使用 `utf8_bin` 排序时。通常Django自身应当顺利地工作使用这样的列（除了 `contrib.sessions` `Session` and `contrib.admin` `LogEntry` 表描述的），如果你想要在你的工作中是数据保持一致，你的代码必须时刻准备调用 `django.utils.encoding.smart_text()` ---Django不会为你做这些（数据库后曾和群体模块层在内部被分离以至于数据库层不知道它需要在这种特殊情况下作出转换）。

如果你使用MySQLdb 1.2.1p2，Django的标准 `CharField` 类型将会返回unicode字符串即使存在 `utf8_bin` 序列。然而, `TextField` 字段将会像一个 `array.array` 实例返回（来自Python的标准 `array` 模块）。Django关于这方面不能做出很多，因为，下一次，当从数据库读取数据时，这些信息需要作出必要转换的行为将不可用。这个问题固定在 MySQLdb 1.2.2，所以如果你想使用使用 `utf8_bin` 序列的 `TextField` ，升级到1.2.2版本然后结合bytestrings处理（这不会太难）如上所述是推荐的解决方案。

你应该决定在 MySQLdb 1.2.1p2 or 1.2.2对你的表使用 `utf8_bin` 排序规则，对于 `django.contrib.sessions.models.Session` 表(通常称为 `django_session`)和 `django.contrib.admin.models.LogEntry` 表 (通常称为 `django_admin_log`)，你仍然应该使用 `utf8_general_ci` (默认)排序规则。Those are the two standard tables that use `TextField` internally.

请注意，根据 MySQL Unicode字符集, 比较下 `utf8_general_ci` 排序规则更加快速，但是与 `utf8_unicode_ci` 比较明显正确率略低。如果在你的应用中这是可接受的,你应该使用 `utf8_general_ci` 因为它更快。如果这是不能接受的 (例如,如果您需要德国字典顺序), 使用 `utf8_unicode_ci` ，因为它更准确。

警告

模块层组以区分大小写的方式验证唯一的字段。因此当使用一个不区分大小写的序列，一个带有唯一字段值的层组，只有通过不同层组（？）才可通过验证，但是依据调用 `save()` ， `IntegrityError` 错误将会被指出。

连接数据库

请参阅 *settings documentation*.

连接设置将被用在这些命令上：

1. `OPTIONS` .
2. `NAME` , `USER` , `PASSWORD` , `HOST` , `PORT`
3. MySQL option files.

换句话说,如果你设置数据库的名称在 `OPTIONS` , 这将优先于 `NAME` , 这将覆盖任何在MySQL option file中的东西.

下面是使用一个MySQL选择文件一个示例配置:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': '/path/to/my.cnf',
        },
    }
}

# my.cnf
[client]
database = NAME
user = USER
password = PASSWORD
default-character-set = utf8
```

其他几个MySQLdb连接选项可能是有用的,例如 `ssl` , `init_command` , and `sql_mode` . 查阅 MySQLdb documentation了解更多细节。

创建你的表

当Django生成数据库得schema（数据库对象的集合）时，它没有指定一个储存引擎，所以你的表将被创建成你服务器配置的默认储存引擎。最简单的解决方案是设置你的数据库服务器的默认存储引擎为你所需求的引擎。

如果你使用托管服务,并且不能改变你的服务器的默认存储引擎,你有两种选择。

- 在表创建之后, 执行 `ALTER TABLE` 语句来转换表为一个新的储存引擎 (例如 InnoDB):

```
ALTER TABLE <tablename> ENGINE=INNODB;
```

```
如果你有很多表，这会很繁琐。
```

- 另一个选择是对于MySQLdb使用 `init_command` 选项在你创建表之前。

```
'OPTIONS': {
        'init_command': 'SET storage_engine=INNODB',
    }
```

```
这设置默认的储存引擎在连接数据库之上。创建了表之后,您应该删除这个选项,因为它
增加了一个只需要在每个表创建数据库连接时的查询。
```

表名

甚至在最新版本的MySQL中，这里存在已知的问题导致当某些SQL语句在一定条件下被执行下，表的名称可以被改变。建议您使用小写的表名，如果可能的话，避免任何可能从这个行为产生的问题。当Django从模型中自动生成表名时，它使用小写的表名，这主要考虑到如果你不顾问题让表名通过 `db_table` 参数。

保存点

Django ORM和MySQL (当使用InnoDB存储引擎) 支持数据库保存点。

如果你使用MyISAM存储引擎请注意，你将会收到数据库生成的错误如果你尝试去使用*savepoint-related methods of the transactions API*。这个原因是检测一个MySQL数据库或表的存储引擎是一项expensive operation，所以它认定为在这些操作结果中无操作的情况下，它不值得去动态转换这些方法。

特定字段说明

Character fields

任何字段都被存储在 `VARCHAR` 列类型的 `max_length` ，它被限制在255个字符之内，如果您对这些字段使用 `unique=True` 的话。 这个影响到 `CharField` , `SlugField` 和 `CommaSeparatedIntegerField` 。

Time and DateTime fields的小数秒的支持

MySQL 5.6.4和更高版本可以存储小数秒，只需要列定义包含一个小数指示 (例如 `DATETIME(6)`).。早期版本不支持它们。此外，MySQLdb的1.2.5以上版本存在一个bug，它会阻止MySQL使用小数秒。

如果数据库服务器支持的话，Django不会升级现有的列去包含小数秒。如果你想确保它们在一个当前的数据库中，这决定由你自己在目标数据库中手动去升级列，通过执行这样的一个命令：

```
ALTER TABLE `your_table` MODIFY `your_datetime_column` DATETIME(
6)
```

或者在数据迁移中使用 `RunSQL` 选项。

Changed in Django 1.8:

以前，Django在使用MySQL后端时删除（truncated）了来自 `datetime` 和 `time` 值的小数秒。现在它让数据库决定他是否应该去删除（drop）部分的值。默认情况下, 新的 `DateTimeField` 或者 `TimeField` 列现在创建小数秒支持在MySQL5.6.4或更高版本和每个mysqlclient或MySQLdb 1.2.5或更高版本。

`TIMESTAMP` 列

如果你使用了包含 `TIMESTAMP` 列的遗留数据库,你必须设置 `USE_TZ = False` 来避免数据丢失。 `inspectdb` 映射这些列在 `DateTimeField` 并且如果你启用时区支持,MySQL和Django将尝试从UTC的值转换为本地时间。

Row locking with `QuerySet.select_for_update()`

MySQL不支持 `NOWAIT` 选项来 `SELECT ...` FOR UPDATE statement. 如果 `select_for_update()` 和 `nowait=True` 一起被使用，会导致一个 `DatabaseError（数据库错误）` 产生。

自动类型转换会导致意想不到的结果

当你执行一个字符串类型的查询时，存在整型值时，MySQL会强制转换表里的所有值变为整型在你进行比较之前。如果你的表里包含 `'abc'` , `'def'` 和你查询 `WHERE mycolumn=0` , 两行相匹配。同样, `WHERE mycolumn=1` 将会匹配 `'abc1'` 。因此，Django中的字符串类型将会总是转换这个值为一个字符串当你查询中使用它之前。

如果你执行直接继承自 `Field` 的自定义模型字段，会覆盖 `get_prep_value()` ，或者使用 `extra()` 或者 `raw()` , 你应该确保执行了适当的类型。

SQLite说明

SQLite 提供了一个极好的开发替代品对于主要为只读或者安装一个小的程序的应用程序。你应该知道的是，如同所有的数据库服务一样，SQLLite有一些不同于其他数据库的差异。

子串匹配和大小写敏感性

对于所有的SQLite版本，存在一些细微的违反语感的行为当你试图匹配某些类型的字符串。当你在Querysets中使用 `iexact` 或者 `contains` 过滤器，这些行为会被触发。 这种行为分为两个情况:

1. 对于子串匹配，所有的匹配都会不区分大小写。这是一个过滤器例如 `filter(name__contains="aa")` 将会匹配一个名称为 `"Aabb"` 。

2. 对于字符串包含ASCII范围之外的字符，所有准确的字符串匹配都会被执行区分大小写的操作，甚至当不区分大小写的选项在查询中通过。所以在这些情况下， `iexact` 过滤器将会表现与 `exact` 过滤器一模一样。

一些可能的解决办法在 documented at sqlite.org，但是他们不能利用默认的SQLite后端在Django中，强行将他们合并将会非常的困难。因此，Django显示默认的SQLite行为，当你使用不区分大小写或者子串过滤操作时你应该意识到这点。

旧版SQLite和 `CASE` 表达式

SQLite 3.6.23.1和更老版本包含了一个bug 当handling query parameters 在一个 `CASE` 表达式中包含了一个 `ELSE` 和算法。

2010年3月SQLite 3.6.23.1发布，目前大多数不同平台的二进制发行版包括一个新版本的SQLite，值得注意的是python 2.7 安装在windows平台上。

在撰写本文时,最新版本Windows - Python 2.7.9包括SQLite 3.6.21。你可以安装 `pysqlite2` 或者替换 `sqlite3.dll` (默认情况下安装在 `C:Python27DLLs`)通过来自http://www.sqlite.org/ 的新版本来补救这个问题。

使用新版本的 SQLite DB-API 2.0驱动

Django将会使用一个 `pysqlite2` 模块优先于附带的 `sqlite3` 如果你发现一个是可用的在Python的标准库中。

这能够升级 DB-API 2.0接口或SQLite 3本身比那些包含在您的特定版本Python二进制发行版更新，如果你需要的话。

“Database is locked” errors

SQLite是一个轻量级的数据库,因此不能支持一个高水平的并发性。 `OperationalError: database is locked` 错误表明你的应用正在经受更高的并发相较于 `sqlite` 能在默认配置中处理的那样。这个错误意味着一个线程或进程在数据库连接上存在一个交互型锁,另一个超时线程等待独自锁被释放。

Python的SQLite封装有一个默认的超时时间值决定了在超时之前第二个线程被允许等待多长时间在lock上并且会反馈 `OperationalError: database is locked` error。

如果你出现了这个错误，你可以这样解决：

- 切换到另一个数据库端。在某种程度上SQLite变得太“lite”对于现实世界的应用程序,并且这些并发错误表明你已经达到了这一点。
- 重写代码,以减少并发性和确保数据库事务是短期的。
- 增加默认超时时间值通过设置 `timeout` 数据库选项：

```
'OPTIONS': {
        # ...
        'timeout': 20,
        # ...
    }
```

```
这只会使SQLite等一段时间才会抛出“database is locked”错误；它不会真的做一些事情去解决他们。
```

`QuerySet.select_for_update()` 不被支持

SQLite不支持 `SELECT ...` FOR UPDATE syntax. 它将没有影响。

“pyformat” parameter style in raw queries not supported

对于大多数的数据库后端, 原始查询(`Manager.raw()` 或者 `cursor.execute()`) 可以使用 “pyformat”参数风格，在查询中placeholders作为 `'%(name)s'` 并且被传递的参数是一个字典而不是列表。SQLite 不支持这个。

在 `connection.queries` 参数不被引用

`sqlite3` 没有提供一种方式来检索SQL引用后和替换的参数。相反，SQL在 `connection.queries` 被一个简单的字符串插值重建。它可能是不正确的。确保你在必要时添加引用钱拷贝一个引用在SQLite shell中。

Oracle说明

Django 支持 Oracle Database Server 11.1和更高版本 。4.3.1或更高版本的 cx_Oracle Python驱动程序是必需的,尽管我们建议5.1.3或更高的支持Python 3的版本。

注意由于一个Unicode-corruption bug 在 `cx_Oracle` 5.0, 这个版本的驱动程序 不能 支持Django使用； `cx_Oracle` 5.0.1解决了这个问题,所以如果你希望使用一个更近时间发布的 `cx_Oracle` , 使用5.0.1版本。

`cx_Oracle` 5.0.1或更高版本支持被随意编辑通过 `WITH_UNICODE` 环境变量。这个建议不是必须的。

为了使 `python manage.py migrate` 命令工作, 您的Oracle数据库用户必须拥有特权运行以下命令:

- CREATE TABLE
- CREATE SEQUENCE
- CREATE PROCEDURE
- CREATE TRIGGER

运行一个项目的测试套件， 用户通常需要这些*additional* 特权：

- CREATE USER
- DROP USER
- CREATE TABLESPACE
- DROP TABLESPACE
- CREATE SESSION WITH ADMIN OPTION
- CREATE TABLE WITH ADMIN OPTION
- CREATE SEQUENCE WITH ADMIN OPTION
- CREATE PROCEDURE WITH ADMIN OPTION
- CREATE TRIGGER WITH ADMIN OPTION

注意, 虽然 RESOURCE role 具有有需求 CREATE TABLE, CREATE SEQUENCE, CREATE PROCEDURE 和 CREATE TRIGGER 特权, 并且一个用户被允许 RESOURCE当ADMIN OPTION能够授权 RESOURCE这样一个用户不能授予个人特权(例如 CREATE TABLE)，因此RESOURCE伴随ADMIN OPTION通畅不满足运行测试。（这段没翻译好。。大家理解一下）

一些测试套件也创建视图;运行这些,用户还需要 CREATE VIEW WITH ADMIN OPTION 特权。特别的是，这是Django自己的测试套件所需要的。

Changed in Django 1.8:

Django 1.8之前,测试用户被允许CONNECT and RESOURCE roles,所以所需的额外的特权运行测试套件是不同的。

所有这些特权都包含在DBA role,这适合使用在一个私人开发人员的数据库。

Oracle数据库后端使用 `SYS.` DBMS_LOB包,所以用户需要执行权限。通常是在默认情况下所有用户都可以访问的,但如果不是,你需要授予权限如下:

```
GRANT EXECUTE ON SYS.DBMS_LOB TO user;
```

连接到数据库

为了连接使用你的Oracle数据库的服务名, 你的 `settings.py` 文件应该像这样：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': '',
        'PORT': '',
    }
}
```

在这种情况下, 你应该让 `HOST` and `PORT` 置空。 然而，如果你不是使用一个 `tnsnames.ora` 文件或者一个类似的命名方法 并且想要连接使用 SID (“xe” 在本例),填写 `HOST` 和 `PORT` 像如下：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': 'dbprod01ned.mycompany.com',
        'PORT': '1540',
    }
}
```

你应该同时填写 `HOST` and `PORT` , 或者全部置空。Django将会取决你的选择使用一个不同的连接描述符。

线程的选项

如果你计划运行Django在多线程环境中(例如Apache使用默认MPM模块在任何现代的操作系统)， 在你的Oracle数据库配置中你 必须 设置 `threaded` 选项为True:

```
'OPTIONS': {
    'threaded': True,
},
```

未能这样做可能会导致崩溃和其他奇怪的行为。

INSERT ... RETURNING INTO

默认情况下,当插入新的行时， Oracle 后端使用 `RETURNING INTO` 去检索一个有效的 `AutoField` 。这个行为可能会导致一个 `DatabaseError` 在某些不寻常设置中, 例如当插入到一个远程表,或者在一个视图中随着一个 `INSTEAD OF` 触

发。 `RETURNING INTO` 子句可以禁用通过 `use_returning_into` 选项在数据库设置中为Flase:

```
'OPTIONS': {
    'use_returning_into': False,
},
```

在这种情况, Oracle后端将会使用一个单独的 `SELECT` 查询来检索AutoField值。

命名问题

Oracle强加了一个30个字符的名称限制。 为了适应, 后端缩短了数据库表示符为最适长度，取代带着一个可重复的MD5散列值的缩短的名称的最后四个字符。此外,后端数据库标识符变成大写。

为了防止这些转换(这通常是只需要在处理遗留数据库或访问表属于其他用户),使用一个引用名称像 `db_table` 值这样:

```
class LegacyModel(models.Model):
    class Meta:
        db_table = '"name_left_in_lowercase"'

class ForeignModel(models.Model):
    class Meta:
        db_table = '"OTHER_USER"."NAME_ONLY_SEEMS_OVER_30"'
```

引用名称也可以被使用在Django的其他受支持的数据库后端;除了r Oracle, 然而,引用没有影响。

当运行 `migrate` ，一个 `ORA-06552` 错误可能出现，如果某些Oracle的keywords被使用作为模型字段名称或一个 `db_column` 选项的值时。Django引用在查询使用的所有的标示符来防止大多数这样的问你，但是这个错误仍然可以发生当Oracle数据类型被用为一个列名。特别是，避免使用这样的名称 `date` , `timestamp` , `number` or `float` 作为一个字段名。

NULL和空字符串

Django通常更喜欢使用空字符串(“”)而不是NULL,但是Oracle将两个相同对待。为了解决这个问题，Oracle后端忽略了字段中一个explict的 `null` 选项，让空字符串作为一个可能的值并生成DDL就像 `null=True` 。当从数据库获取时,它假定一个 `NULL` 值在这些字段之一意味一个空字符串，并且数据默认转换去反映这种假设。

`TextField` 限制

The Oracle后端存储 `TextFields` 像 `NCLOB` 列.。通常使用LOB列时，Oracle 强加了一些限制：

- LOB 列不被像主键一样使用。
- LOB列不得用于索引。
- LOB列不被使用在一个 `SELECT DISTINCT` 列表。 这意味着当试图去使用 `QuerySet.distinct` 方法在一个包含 `TextField` 列的模型中将会导致一个错误当违反Oracle时。一个解决方案是, 使用 `QuerySet.defer` 方法连同 `distinct()` 来防止将要包含在 `SELECT DISTINCT` 列表中的 `TextField` 列。

使用第三方数据库后端

除了官方支持的数据库,这些第三方提供的后端也允许您在Django中使用其他数据库：

- SAP SQL Anywhere
- IBM DB2
- Microsoft SQL Server
- Firebird
- ODBC
- ADSDB

Django版本和ORM特性被支持通过这些非官方的后端相当大的变化。查询这些非官方的后端有关的特定功能，以及任何支持的查询，应该针对每个第三方项目提供的支持项目。

将遗留数据库整合到Django

虽然Django最适合用来开发新的应用，但也可以将它整合到遗留的数据库中。Django包含了很多工具，尽可能自动化解决这类问题。

这篇文章假设你了解Django的基础部分，它们在教程中提及。

一旦你的Django环境建立好之后，你可以按照这个大致的流程，整合你的现有数据库。

向Django提供你的数据库参数

你需要告诉Django你的数据库连接参数，以及数据库的名称。请修改DATABASES设置，为'默认' 连接的以下键赋值：

- NAME
- ENGINE
- USER
- PASSWORD
- HOST
- PORT

自动生成模型

Django自带叫做inspectdb的工具，可以按照现有的数据库创建模型。你可以运行以下命令，并查看输出：

```
$ python manage.py inspectdb
```

通过重定向Unix标准输出流来保存文件：

```
$ python manage.py inspectdb > models.py
```

这个特性是一个快捷方式，并不是一个确定的模型生成器。详见inspectdb文档 。

一旦你创建好了你的模型，把文件命名为models.py，然后把它放到你应用的Python包中。然后把应用添加到你的INSTALLED_APPS 设置中。

默认情况下，inspectdb创建未被管理的模型。这就是说，模型的Meta类中的managed = False告诉Django不要管理每个表的创建、修改和删除：

```
class Person(models.Model):
    id = models.IntegerField(primary_key=True)
    first_name = models.CharField(max_length=70)
    class Meta:
       managed = False
       db_table = 'CENSUS_PERSONS'
```

如果你希望Django管理表的生命周期，你需要把managed选项改为 True（或者简单地把它移除，因为True是默认值）。

安装Django核心表

接下来，运行migrate命令来安装所有所需的额外的数据库记录，比如后台权限和内容类型：

```
$ python manage.py migrate
```

测试和调整

上面就是所有基本的步骤了 —— 到目前为止你会想要调整Django自动生成的模型，直到他们按照你想要的方式工作。尝试通过Django数据库API访问你的数据，并且尝试使用Django后台页面编辑对象，以及相应地编辑模型文件。

为模型提供初始数据

当你首次建立一个应用的时候，为你的数据库预先安装一些硬编码的数据，是很有用处的。 有几种方法可以让Django自动创建这些数据：你可以通过fixtures提供初始数据，或者提供一个包含初始数据的sql文件。

通常来讲，使用fixtrue更加简洁，因为它是数据库无关的，而使用sql初始化更加灵活。

提供初始数据的fixtures

fixture是数据的集合，让Django了解如何导入到数据库中。创建fixture的最直接的方式，是使用manage.py dumpdata命令，如果数据库中已经有了一些数据。或者你可以手写fixtures。fixtures支持JSON、XML或者YAML（需要安装PyYAML）文档。序列化文档中详细阐述了每一种所支持的序列化格式。

下面这个例子展示了一个简单的Person 模型的fixtrue，看起来很像JSON：

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

下面是它的YAML格式：

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

你可以把这些数据储存在你应用的fixtures目录中。

加载数据很简单：只要调用manage.py loaddata <fixturename>就好了，其中<fixturename>是你所创建的fixture文件的名字。每次你运行loaddata的时候，数据都会从fixture读出，并且重复加载进数据库。注意这意味着，如果你修改了fixtrue创建的某一行，然后再次运行了 loaddata，你的修改将会被抹掉。

自动加载初始数据的fixtures

```
1.7中废除：

如果一个应用使用了迁移，将不会自动加载fixtures。由于Django 1.9中，迁移将
会是必要的，这一行为经权衡之后被废除。 如果你想在一个应用中加载初始数据，考虑
在数据迁移中加载它们。
```

如果你创建了一个命名为 initial_data.[xml/yaml/json]的fixtrue，在你每次运行migrate命令时，fixtrue都会被加载。这非常方面，但是要注意：记住数据在你每次运行migrate命令后都会被刷新。So don’t use initial_data for data you’ll want to edit.

Django在哪里寻找fixture文件

通常，Django 在每个应用的fixtures目录中寻找fixture文件。你可以设置FIXTURE_DIRS选项为一个额外目录的列表，Django会从里面寻找。

运行manage.py loaddata命令的时候，你也可以指定一个fixture文件的目录，它会覆盖默认设置中的目录。

> 另见
>
> fixtrues也被用于测试框架来搭建一致性的测试环境。

提供初始SQL数据

```
1.7中废除：

如果一个应用使用迁移，初始SQL数据将不会加载（包括后端特定的SQL数据）。由于D
jango 1.9中，迁移将会是必须的，这一行为经权衡后被废除。如果你想在应用中使用
初始SQL数据，考虑在数据迁移中使用它们。
```

Django为数据库无关的SQL提供了一个钩子，当你运行migrate命令时，CREATE TABLE语句执行之后就会执行它。你可以使用这个钩子来建立默认的记录，或者创建SQL函数、视图、触发器以及其它。

钩子十分简单：Django会在你应用的目录中寻找叫做sql/<modelname>.sql的文件，其中 <modelname>是小写的模型名称。

所以如果在myapp应用中存在Person模型，你应该在myapp目录的文件sql/person.sql中添加数据库无关的SQL。下面的例子展示了文件可能会包含什么：

```
INSERT INTO myapp_person (first_name, last_name) VALUES ('John', 'Lennon');
INSERT INTO myapp_person (first_name, last_name) VALUES ('Paul', 'McCartney');
```

每个提供的SQL文件，都应该含有用于插入数据的有效的SQL语句（例如，格式适当的INSERT语句，用分号分隔）。

这些SQL文件可被manage.py中的 sqlcustom和sqlall命令阅读。详见manage.py文档。

注意如果你有很多SQL数据文件，他们执行的顺序是不确定的。唯一可以确定的是，在你的自定义数据文件被执行之前，所有数据表都被创建好了。

> 初始SQL数据和测试
>
> 这一技巧不能以测试目的用于提供初始数据。Django的测试框架在每次测试后都会刷新测试数据库的内容。所以，任何使用自定义SQL钩子添加的数据都会丢失。
>
> 如果你需要在测试用例中添加数据，你应该在测试fixture中添加它，或者在测试用例的setUp()中添加。

数据库后端特定的SQL数据

没有钩子提供给后端特定的SQL数据。例如，你有分别为PostgreSQL和SQLite准备的初始数据文件。对于每个应用，Django都会寻找叫做<app_label>/sql/<modelname>.<backend>.sql的文件，其中<app_label>是小写的模型名称，<modelname>是小写的模型名称，<backend>是你的设置文件中由

ENGINE提供的模块名称的最后一部分（例如，如果你定义了一个数据库，ENGINE的值为django.db.backends.sqlite3，Django会寻找<app_label>/sql/<modelname>.sqlite3.sql）。

后端特定的SQL数据会先于后端无关的SQL数据执行。例如，如果你的应用包含了sql/person.sql 和sql/person.sqlite3.sql文件，而且你已经安装了SQLite应用，Django会首先执行 sql/person.sqlite3.sql的内容，其次才是sql/person.sql。

数据库访问优化

Django的数据库层提供了很多方法来帮助开发者充分的利用他们的数据库。这篇文档收集了相关文档的一些链接，添加了大量提示，并且按照优化数据库使用的步骤的概要来组织。

性能优先

作为通用的编程实践，性能的重要性不用多说。弄清楚你在执行什么查询以及你的开销花在哪里。你也可能想使用外部的项目，像django-debug-toolbar，或者直接监控数据库的工具。

记住你可以优化速度、内存占用，甚至二者一起，这取决于你的需求。一些针对其中一个的优化会对另一个不利，但有时会对二者都有帮助。另外，数据库进程做的工作，可能和你在Python代码中做的相同工作不具有相同的开销。决定你的优先级是什么，是你自己的事情，你必须要权衡利弊，按需使用它们，因为这取决于你的应用和服务器。

对于下面提到的任何事情，要记住在任何修改后验证一下，确保修改是有利的，并且足够有利，能超过你代码中可读性的下降。下面的所有建议都带有警告，在你的环境中大体原则可能并不适用，或者会起到相反的效果。

使用标准数据库优化技巧

...包括：

- 索引。在你决定哪些索引应该添加 之后，这一条具有最高优先级。使用Field.db_index或者Meta.index_together在Dhango中添加它们。考虑在你经常使用filter()、exclude()、order_by()和其它方法查询的字段上面添加索引，因为索引有助于加速查找。注意，设计最好的索引方案是一个复杂的、数据库相关的话题，它取决于你应用的细节。持有索引的副作用可能会超过查询速度上的任何收益。
- 合理使用字段类型。

我们假设你已经完成了上面这些显而易见的事情。这篇文档剩下的部分，着重于讲解如何以不做无用功的方式使用Django。这篇文档也没有强调用在开销大的操作上其它的优化技巧，像general purpose caching。

理解查询集

理解查询集(QuerySets) 是通过简单的代码获取较好性能至关重要的一步。特别是：

理解查询集计算

要避免性能问题，理解以下几点非常重要：

- QuerySets是延迟的。
- 什么时候它们被计算出来。
- 数据在内存中如何存储。

理解缓存属性

和整个QuerySet的缓存相同，ORM对象的属性的结果中也存在缓存。通常来说，不可调用的属性会被缓存。例如下面的博客模型示例：

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog   # Blog object is retrieved at this point
>>> entry.blog   # cached version, no DB access
```

但是通常来讲，可调用的属性每一次都会访问数据库。

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()   # query performed
>>> entry.authors.all()   # query performed again
```

要小心当你阅读模板代码的时候 —— 模板系统不允许使用圆括号，但是会自动调用callable对象，会隐藏上述区别。

要小心使用你自定义的属性 —— 实现所需的缓存取决于你，例如使用cached_property装饰符。

使用**with**模板标签

要利用QuerySet的缓存行为，你或许需要使用with模板标签。

使用**iterator()**

当你有很多对象时，QuerySet的缓存行为会占用大量的内存。这种情况下，采用iterator()解决。

在数据库中而不是**Python**中做数据库的工作

比如：

- 在最基础的层面上，使用过滤器和反向过滤器对数据库进行过滤。
- 使用F 表达式在相同模型中基于其他字段进行过滤。

- 使用数据库中的注解和聚合。

如果上面那些都不够用，你可以自己生成SQL语句：

使用QuerySet.extra()

extra()是一个移植性更差，但是功能更强的方法，它允许一些SQL语句显式添加到查询中。如果这些还不够强大：

使用原始的SQL

编写你自己的自定义SQL语句，来获取数据或者填充模型。使用django.db.connection.queries来了解Django为你编写了什么，以及从这里开始。

用唯一的被或索引的列来检索独立对象

有两个原因在get()中，用带有unique或者db_index的列检索独立对象。首先，由于查询经过了数据库的索引，所以会更快。其次，如果很多对象匹配查询，查询会更慢一些；列上的唯一性约束确保这种情况永远不会发生。

所以，使用博客模型的例子：

```
>>> entry = Entry.objects.get(id=10)
```

会快于：

```
>>> entry = Entry.object.get(headline="News Item Title")
```

因为id被数据库索引，而且是唯一的。

下面这样做会十分缓慢：

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

首先， headline没有被索引，它会使查询变得很慢：

其次，这次查找并不确保返回唯一的对象。如果查询匹配到多于一个对象，它会在数据库中遍历和检索所有这些对象。如果记录中返回了成百上千个对象，代价是非常大的。如果数据库运行在分布式服务器上，网络开销和延迟也是一大因素，代价会是它们的组合。

一次性检索你需要的任何东西

在不同的位置多次访问数据库，一次获取一个数据集，通常来说不如在一次查询中获取它们更高效。如果你在一个循环中执行查询，这尤其重要。有可能你会做很多次数据库查询，但只需要一次就够了。所以：

使用QuerySet.select_related()和prefetch_related()

充分了解并使用select_related()和prefetch_related()：

- 在视图的代码中，
- 以及在适当的管理器和默认管理器中。要意识到你的管理器什么时候被使用和不被使用；有时这很复杂，所以不要有任何假设。

不要获取你不需要的东西

使用QuerySet.values()和values_list()

当你仅仅想要一个带有值的字典或者列表，并不需要使用ORM模型对象时，可以适当使用values()。对于在模板代码中替换模型对象，这样会非常有用 —— 只要字典中带有的属性和模板中使用的一致，就没问题。

使用QuerySet.defer()和only()

如果一些数据库的列你并不需要（或者大多数情况下并不需要），使用defer()和only()来避免加载它们。注意如果你确实要用到它们，ORM会在另外的查询之中获取它们。如果你不能够合理地使用这些函数，不如不用。

另外，当建立起一个带有延迟字段的模型时，要意识到一些（小的、额外的）消耗会在Django内部产生。不要不分析数据库就盲目使用延迟字段，因为数据库必须从磁盘中读取大多数非text和VARCHAR数据，在结果中作为单独的一行，即使其中的列很少。 defer()和only()方法在你可以避免加载大量文本数据，或者可能要花大量时间处理而返回给Python的字段时，特别有帮助。像往常一样，应该先写出个大概，之后再优化。

使用QuerySet.count()

...如果你想要获取大小，不要使用 len(queryset)。

使用QuerySet.exists()

...如果你想要知道是否存在至少一个结果，不要使用if queryset。

但是：

不要过度使用 count() 和 exists()

如果你需要查询集中的其他数据，就把它加载出来。

例如，假设Email模型有一个body属性，并且和User有多对多的关联，下面的的模板代码是最优的：

```
{% if display_inbox %}
  {% with emails=user.emails.all %}
    {% if emails %}
      <p>You have {{ emails|length }} email(s)</p>
      {% for email in emails %}
        <p>{{ email.body }}</p>
      {% endfor %}
    {% else %}
      <p>No messages today.</p>
    {% endif %}
  {% endwith %}
{% endif %}
```

这是因为：

- 因为查询集是延迟加载的，如果'display_inbox'为False，不会查询数据库。
- 使用with意味着我们为了以后的使用，把user.emails.all储存在一个变量中，允许它的缓存被复用。
- {% if emails %}的那一行调用了QuerySet.**bool**()，它导致user.emails.all()查询在数据库上执行，并且至少在第一行以一个ORM对象的形式返回。如果没有任何结果，会返回False，反之为True。
- {{ emails|length }}调用了QuerySet.**len**()方法，填充了缓存的剩余部分，而且并没有执行另一次查询。
- for循环的迭代器访问了已经缓存的数据。

总之，这段代码做了零或一次查询。唯一一个慎重的优化就是with标签的使用。在任何位置使用QuerySet.exists()或者QuerySet.count()都会导致额外的查询。

使用**QuerySet.update()**和**delete()**

通过QuerySet.update()使用批量的SQL UPDATE语句，而不是获取大量对象，设置一些值再单独保存。与此相似，在可能的地方使用批量deletes。

但是要注意，这些批量的更新方法不会在单独的实例上面调用save()或者delete()方法，意思是任何你向这些方法添加的自定义行为都不会被执行，包括由普通数据库对象的信号驱动的任何方法。

直接使用外键的值

如果你仅仅需要外键当中的一个值，要使用对象上你已经取得的外键的值，而不是获取整个关联对象再得到它的主键。例如，执行：

```
entry.blog_id
```

而不是：

```
entry.blog.id
```

不要做无谓的排序

排序并不是没有代价的；每个需要排序的字段都是数据库必须执行的操作。如果一个模型具有默认的顺序（Meta.ordering），并且你并不需要它，通过在查询集上无参调用order_by() 来移除它。

向你的数据库添加索引可能有助于提升排序性能。

整体插入

创建对象时，尽可能使用bulk_create()来减少SQL查询的数量。例如：

```
Entry.objects.bulk_create([
    Entry(headline="Python 3.0 Released"),
    Entry(headline="Python 3.1 Planned")
])
```

...更优于：

```
Entry.objects.create(headline="Python 3.0 Released")
Entry.objects.create(headline="Python 3.1 Planned")
```

注意该方法有很多注意事项，所以确保它适用于你的情况。

这也可以用在ManyToManyFields中，所以：

```
my_band.members.add(me, my_friend)
```

...更优于：

```
my_band.members.add(me)
my_band.members.add(my_friend)
```

...其中Bands和Artists具有多对多关联。

视图层

Django 具有“视图”的概览，用于封装负责处理用户请求及返回响应的逻辑。通过下面的链接可以找到你需要知道的所有关于视图的内容：

基础

URL调度器

简洁、优雅的URL 模式在高质量的Web 应用中是一个非常重要的细节。Django 允许你任意设计你的URL，不受框架束缚。

不要求有 `.php` 或 `.cgi` ，更不会要求类似 `0,2097,1-1-1928,00` 这样无意义的东西。

参见万维网的发明者Berners-Lee 的Cool URIs don't change，里面有关于为什么URL 应该保持整洁和有意义的卓越的论证。

概览

为了给一个应用设计URL，你需要创建一个Python 模块，通常称为URLconf（URL configuration）。这个模块是纯粹的Python 代码，包含URL 模式（简单的正则表达式）到Python 函数（你的视图）的简单映射。

映射可短可长，随便你。它可以引用其它的映射。而且，因为它是纯粹的Python 代码，它可以动态构造。

Django 还提供根据当前语言翻译URL 的一种方法。更多信息参见国际化文档。

Django 如何处理一个请求

当一个用户请求Django 站点的一个页面，下面是Django 系统决定执行哪个Python 代码使用的算法：

1. Django 决定要使用的根 `URLconf` 模块。通常，这个值就是 `ROOT_URLCONF` 的设置，但是如果进来的 `HttpRequest` 对象具有一个 `urlconf` 属性（通过中间件 `request` `processing` 设置），则使用这个值来替换 `ROOT_URLCONF` 设置。
2. Django 加载该Python 模块并寻找可用的 `urlpatterns` 。它是 `django.conf.urls.url()` 实例的一个Python 列表。
3. Django 依次匹配每个URL 模式，在与请求的URL 匹配的第一个模式停下来。
4. 一旦其中的一个正则表达式匹配上，Django 将导入并调用给出的视图，它是一个简单的Python 函数（或者一个基于类的视图）。视图将获得如下参数:
    - 一个 `HttpRequest` 实例。
    - 如果匹配的正则表达式没有返回命名的组，那么正则表达式匹配的内容将作为位置参数提供给视图。
    - 关键字参数由正则表达式匹配的命名组组成，但是可以被 `django.conf.urls.url()` 的可选参数 `kwargs` 覆盖。
5. 如果没有匹配到正则表达式，或者如果过程中抛出一个异常，Django 将调用一个适当的错误处理视图。请参见下面的错误处理。

例子

下面是一个简单的 URLconf：

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archiv
e),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.art
icle_detail),
]
```

注：

- 若要从URL 中捕获一个值，只需要在它周围放置一对圆括号。
- 不需要添加一个前导的反斜杠，因为每个URL 都有。例如，应该是 `^articles` 而不是 `^/articles` 。
- 每个正则表达式前面的'r' 是可选的但是建议加上。它告诉Python 这个字符串是"原始的" —— 字符串中任何字符都不应该转义。参见Dive Into Python 中的解释。

一些请求的例子：

- `/articles/2005/03/` 请求将匹配列表中的第三个模式。Django 将调用函数 `views.month_archive(request, '2005', '03')` 。
- `/articles/2005/3/` 不匹配任何URL 模式，因为列表中的第三个模式要求月份应该是两个数字。
- `/articles/2003/` 将匹配列表中的第一个模式不是第二个，因为模式按顺序匹配，第一个会首先测试是否匹配。请像这样自由插入一些特殊的情况来探测匹配的次序。
- `/articles/2003` 不匹配任何一个模式，因为每个模式要求URL 以一个反斜线结尾。
- `/articles/2003/03/03/` 将匹配最后一个模式。Django 将调用函数 `views.article_detail(request, '2003', '03', '03')` 。

命名组

上面的示例使用简单的、没有命名的正则表达式组（通过圆括号）来捕获URL 中的值并以位置 参数传递给视图。在更高级的用法中，可以使用命名的正则表达式组来捕获URL 中的值并以关键字 参数传递给视图。

在Python 正则表达式中，命名正则表达式组的语法是 (?P<name>pattern) ，其中 name 是组的名称， pattern 是要匹配的模式。

下面是以上URLconf 使用命名组的重写：

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', v
iews.month_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<d
ay>[0-9]{2})/$', views.article_detail),
]
```

这个实现与前面的示例完全相同，只有一个细微的差别：捕获的值作为关键字参数而不是位置参数传递给视图函数。例如：

- `/articles/2005/03/` 请求将调用 `views.month_archive(request, year='2005', month='03')` 函数，而不是 `views.month_archive(request, '2005', '03')` 。
- `/articles/2003/03/03/` 请求将调用函数 `views.article_detail(request, year='2003', month='03', day='03` 。

在实际应用中，这意味你的URLconf 会更加明晰且不容易产生参数顺序问题的错误 —— 你可以在你的视图函数定义中重新安排参数的顺序。当然，这些好处是以简洁为代价；有些开发人员认为命名组语法丑陋而繁琐。

匹配/分组算法

下面是URLconf 解析器使用的算法，针对正则表达式中的命名组和非命名组：

1. 如果有命名参数，则使用这些命名参数，忽略非命名参数。
2. 否则，它将以位置参数传递所有的非命名参数。

根据传递额外的选项给视图函数（下文），这两种情况下，多余的关键字参数也将传递给视图。

URLconf 在什么上查找

URLconf 在请求的URL 上查找，将它当做一个普通的Python 字符串。不包括GET和POST参数以及域名。

例如，http://www.example.com/myapp/ 请求中，URLconf 将查找 `myapp/` 。

在 http://www.example.com/myapp/?page=3 请求中，URLconf 仍将查找 `myapp/` 。

URLconf 不检查请求的方法。换句话讲，所有的请求方法 —— 同一个URL的 `POST` 、 `GET` 、 `HEAD` 等等 —— 都将路由到相同的函数。

捕获的参数永远是字符串

每个捕获的参数都作为一个普通的Python 字符串传递给视图，无论正则表达式使用的是什么匹配方式。例如，下面这行URLconf 中：

```
url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
```

... `views.year_archive()` 的 `year` 参数将是一个字符串，即使 `[0-9]{4}` 值匹配整数字符串。

指定视图参数的默认值

有一个方便的小技巧是指定视图参数的默认值。 下面是一个URLconf 和视图的示例：

```
# URLconf
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^blog/$', views.page),
    url(r'^blog/page(?P<num>[0-9]+)/$', views.page),
]

# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to
num.
    ...
```

在上面的例子中，两个URL模式指向同一个视图 `views.page` —— 但是第一个模式不会从URL 中捕获任何值。如果第一个模式匹配， `page()` 函数将使用 `num` 参数的默认值"1"。如果第二个模式匹配， `page()` 将使用正则表达式捕获的 `num` 值。

性能

`urlpatterns` 中的每个正则表达式在第一次访问它们时被编译。这使得系统相当快。

urlpatterns 变量的语法

`urlpatterns` 应该是 `url()` 实例的一个Python 列表。

错误处理

当Django 找不到一个匹配请求的URL 的正则表达式时，或者当抛出一个异常时，Django 将调用一个错误处理视图。

这些情况发生时使用的视图通过4个变量指定。它们的默认值应该满足大部分项目，但是通过赋值给它们以进一步的自定义也是可以的。

完整的细节请参见自定义错误视图。

这些值可以在你的根URLconf 中设置。在其它URLconf 中设置这些变量将不会生效果。

它们的值必须是可调用的或者是表示视图的Python 完整导入路径的字符串，可以方便地调用它们来处理错误情况。

这些值是：

- `handler404` —— 参见 `django.conf.urls.handler404` 。
- `handler500` —— 参见 `django.conf.urls.handler500` 。
- `handler403` —— 参见 `django.conf.urls.handler403` 。
- `handler400` —— 参见 `django.conf.urls.handler400` 。

包含其它的URLconfs

在任何时候，你的urlpatterns 都可以包含其它URLconf 模块。这实际上将一部分URL 放置与其它URL 下面。

例如，下面是URLconf for the Django 网站自己的URLconf 中一个片段。它包含许多其它URLconf：

```
from django.conf.urls import include, url

urlpatterns = [
    # ... snip ...
    url(r'^community/', include('django_website.aggregator.urls'
)),
    url(r'^contact/', include('django_website.contact.urls')),
    # ... snip ...
]
```

注意，这个例子中的正则表达式没有包含$（字符串结束匹配符），但是包含一个末尾的反斜杠。每当Django 遇到 `include()` （ `django.conf.urls.include()` ）时，它会去掉URL 中匹配的部分并将剩下的字符串发送给包含的URLconf 做进一步处理。

另外一种包含其它URL 模式的方式是使用一个url() 实例的列表。例如，请看下面的URLconf：

```
from django.conf.urls import include, url

from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    url(r'^reports/(?P<id>[0-9]+)/$', credit_views.report),
    url(r'^charge/$', credit_views.charge),
]

urlpatterns = [
    url(r'^$', main_views.homepage),
    url(r'^help/', include('apps.help.urls')),
    url(r'^credit/', include(extra_patterns)),
]
```

在这个例子中， `/credit/reports/` URL将被 `credit.views.report()` 这个Django 视图处理。

这种方法可以用来去除URLconf 中的冗余，其中某个模式前缀被重复使用。例如，考虑这个URLconf：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/history/$', vi
ews.history),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/edit/$', views
.edit),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/discuss/$', vi
ews.discuss),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/permissions/$'
, views.permissions),
]
```

我们可以改进它，通过只声明共同的路径前缀一次并将后面的部分分组：

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/', include([
        url(r'^history/$', views.history),
        url(r'^edit/$', views.edit),
        url(r'^discuss/$', views.discuss),
        url(r'^permissions/$', views.permissions),
    ])),
]
```

捕获的参数

被包含的URLconf 会收到来之父URLconf 捕获的任何参数，所以下面的例子是合法的：

```
# In settings/urls/main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
]

# In foo/urls/blog.py
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.blog.index),
    url(r'^archive/$', views.blog.archive),
]
```

在上面的例子中，捕获的" `username` "变量将被如期传递给包含的 URLconf。

嵌套的参数

正则表达式允许嵌套的参数，Django 将解析它们并传递给视图。当反查时，Django 将尝试填满所有外围捕获的参数，并忽略嵌套捕获的参数。考虑下面的 URL 模式，它带有一个可选的 `page` 参数：

```
from django.conf.urls import url

urlpatterns = [
    url(r'blog/(page-(\d+)/)?$', blog_articles),
  # bad
    url(r'comments/(?:page-(?P<page_number>\d+)/)?$', comments),
  # good
]
```

两个模式都使用嵌套的参数，其解析方式是：例如 `blog/page-2/` 将匹配 `blog_articles` 并带有两个位置参数 `page-2/` 和2。第二个 `comments` 的模式将匹配 `comments/page-2/` 并带有一个值为2 的关键字参数 `page_number` 。这个例子中外围参数是一个不捕获的参数 `(?:...)` 。

`blog_articles` 视图需要最外层捕获的参数来反查，在这个例子中是 `page-2/` 或者没有参数，而 `comments` 可以不带参数或者用一个 `page_number` 值来反查。

嵌套捕获的参数使得视图参数和URL 之间存在强耦合，正如 `blog_articles` 所示：视图接收URL（ `page-2/` ）的一部分，而不只是视图感兴趣的值。这种耦合在反查时更加显著，因为反查视图时我们需要传递URL 的一个片段而不只是page 的值。

作为一个经验的法则，当正则表达式需要一个参数但视图忽略它的时候，只捕获视图需要的值并使用非捕获参数。

传递额外的选项给视图函数

URLconfs 具有一个钩子，让你传递一个Python 字典作为额外的参数传递给视图函数。

`django.conf.urls.url()` 函数可以接收一个可选的第三个参数，它是一个字典，表示想要传递给视图函数的额外关键字参数。

例如：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

在这个例子中，对于 `/blog/2005/` 请求，Django 将调用 `views.year_archive(request, year='2005', foo='bar')` 。

这个技术在Syndication 框架 中使用，来传递元数据和选项给视图。

> 处理冲突
>
> URL 模式捕获的命名关键字参数和在字典中传递的额外参数有可能具有相同的名称。当这种情况发生时，将使用字典中的参数而不是URL 中捕获的参数。

传递额外的选项给include()

类似地，你可以传递额外的选项给include()。当你传递额外的选项给include() 时，被包含的URLconf 的每一 行将被传递这些额外的选项。

例如，下面两个URLconf 设置功能上完全相同：

设置一次：

```
# main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^blog/', include('inner'), {'blogid': 3}),
]

# inner.py
from django.conf.urls import url
from mysite import views

urlpatterns = [
    url(r'^archive/$', views.archive),
    url(r'^about/$', views.about),
]
```

设置两次：

```
# main.py
from django.conf.urls import include, url
from mysite import views

urlpatterns = [
    url(r'^blog/', include('inner')),
]

# inner.py
from django.conf.urls import url

urlpatterns = [
    url(r'^archive/$', views.archive, {'blogid': 3}),
    url(r'^about/$', views.about, {'blogid': 3}),
]
```

注意，额外的选项将永远传递给被包含的URLconf 中的每一行，无论该行的视图实际上是否认为这些选项是合法的。由于这个原因，该技术只有当你确定被包含的URLconf 中的每个视图都接收你传递给它们的额外的选项。

URL 的反向解析

在使用Django 项目时，一个常见的需求是获得URL 的最终形式，以用于嵌入到生成的内容中（视图中和显示给用户的URL等）或者用于处理服务器端的导航（重定向等）。

人们强烈希望不要硬编码这些URL（费力、不可扩展且容易产生错误）或者设计一种与URLconf 毫不相关的专门的URL 生成机制，因为这样容易导致一定程度上产生过期的URL。

换句话讲，需要的是一个DRY 机制。除了其它有点，它还允许设计的URL 可以自动更新而不用遍历项目的源代码来搜索并替换过期的URL。

获取一个URL 最开始想到的信息是处理它视图的标识（例如名字），查找正确的URL 的其它必要的信息有视图参数的类型（位置参数、关键字参数）和值。

Django 提供一个办法是让URL 映射是URL 设计唯一的地方。你填充你的URLconf，然后可以双向使用它：

- 根据用户/浏览器发起的URL 请求，它调用正确的Django 视图，并从URL 中提取它的参数需要的值。
- 根据Django 视图的标识和将要传递给它的参数的值，获取与之关联的URL。

第一种方式是我们在前面的章节中一直讨论的用法。第二种方式叫做反向解析URL、反向URL 匹配、反向URL 查询或者简单的URL 反查。

在需要URL 的地方，对于不同层级，Django 提供不同的工具用于URL 反查：

- 在模板中：使用url 模板标签。
- 在Python 代码中：使用 `django.core.urlresolvers.reverse()` 函数。
- 在更高层的与处理Django 模型实例相关的代码中：使用 `get_absolute_url()` 方法。

例子

考虑下面的URLconf：

```
from django.conf.urls import url

from . import views

urlpatterns = [
    #...
    url(r'^articles/([0-9]{4})/$', views.year_archive, name='new
s-year-archive'),
    #...
]
```

根据这里的设计，某一年nnnn对应的归档的URL是 `/articles/nnnn/` 。

你可以在模板的代码中使用下面的方法获得它们：

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>

<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }
} Archive</a></li>
{% endfor %}
</ul>
```

在Python 代码中，这样使用：

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redirect_to_year(request):
    # ...
    year = 2006
    # ...
    return HttpResponseRedirect(reverse('news-year-archive', arg
s=(year,)))
```

如果出于某种原因决定按年归档文章发布的URL应该调整一下，那么你将只需要修改URLconf 中的内容。

在某些场景中，一个视图是通用的，所以在URL 和视图之间存在多对一的关系。对于这些情况，当反查URL 时，只有视图的名字还不够。请阅读下一节来了解Django 为这个问题提供的解决办法。

命名URL 模式

为了完成上面例子中的URL 反查，你将需要使用命名的URL 模式。URL 的名称使用的字符串可以包含任何你喜欢的字符。不只限制在合法的Python 名称。

当命名你的URL 模式时，请确保使用的名称不会与其它应用中名称冲突。如果你的URL 模式叫做 `comment` ，而另外一个应用中也有一个同样的名称，当你在模板中使用这个名称的时候不能保证将插入哪个URL。

在URL 名称中加上一个前缀，比如应用的名称，将减少冲突的可能。我们建议使用 `myapp-comment` 而不是 `comment` 。

URL 命名空间

简介

URL 命名空间允许你反查到唯一的命名URL 模式，即使不同的应用使用相同的URL 名称。第三方应用始终使用带命名空间的URL 是一个很好的实践（我们在教程中也是这么做的）。类似地，它还允许你在一个应用有多个实例部署的情况下反查URL。换句话讲，因为一个应用的多个实例共享相同的命名URL，命名空间将提供一种区分这些命名URL 的方法。

在一个站点上，正确使用URL 命名空间的Django 应用可以部署多次。例如， `django.contrib.admin` 具有一个 `AdminSite` 类，它允许你很容易地部署多个管理站点的实例。在下面的例子中，我们将讨论在两个不同的地方部署教程中的polls 应用，这样我们可以为两种不同的用户（作者和发布者）提供相同的功能。

一个URL 命名空间有两个部分，它们都是字符串：

应用命名空间

它表示正在部署的应用的名称。一个应用的每个实例具有相同的应用命名空间。例如，可以预见Django 的管理站点的应用命名空间是' `admin` '。

实例命名空间

它表示应用的一个特定的实例。实例的命名空间在你的全部项目中应该是唯一的。但是，一个实例的命名空间可以和应用的命名空间相同。它用于表示一个应用的默认实例。例如，Django 管理站点实例具有一个默认的实例命名空间'admin'。 URL 的命名空间使用':' 操作符指定。例如，管理站点应用的主页使用' `admin:index` '。它表示' `admin` ' 的一个命名空间和' `index` ' 的一个命名URL。

命名空间也可以嵌套。命名URL' `sports:polls:index` ' 将在命名空间' `polls` '中查找' `index` '，而 `poll` 定义在顶层的命名空间' `sports` ' 中。

反查带命名空间的URL

当解析一个带命名空间的URL（例如' `polls:index` '）时，Django 将切分名称为多个部分，然后按下面的步骤查找：

1. 首先，Django 查找匹配的应用的命名空间(在这个例子中为' `polls` '）。这将得到该应用实例的一个列表。
2. 如果有定义当前 应用，Django 将查找并返回那个实例的URL 解析器。当前 应用可以通过请求上的一个属性指定。希望可以多次部署的应用应该设置正在处理的 `request` 上的 `current_app` 属性。

```
Changed in Django 1.8:

在以前版本的Django 中，你必须在用于渲染模板的每个`Context` 或 `RequestC
ontext`上设置`current_app` 属性。
```

当前应用还可以通过 `reverse()` 函数的一个参数手工设定。

1. 如果没有当前应用。Django 将查找一个默认的应用实例。默认的应用实例是实例命名空间 与应用命名空间 一致的那个实例（在这个例子中， `polls` 的一个叫做' `polls` ' 的实例）。
2. 如果没有默认的应用实例，Django 将该应用挑选最后部署的实例，不管实例的名称是什么。
3. 如果提供的命名空间与第1步中的应用命名空间 不匹配，Django 将尝试直接将此命名空间作为一个实例命名空间查找。

如果有嵌套的命名空间，将为命名空间的每个部分重复调用这些步骤直至剩下视图的名称还未解析。然后该视图的名称将被解析到找到的这个命名空间中的一个URL。

例子

为了演示解析的策略，考虑教程中polls 应用的两个实例：' `author-polls` ' 和' `publisher-polls` '。假设我们已经增强了该应用，在创建和显示投票时考虑了实例命名空间。

```
#urls.py

from django.conf.urls import include, url

urlpatterns = [
    url(r'^author-polls/', include('polls.urls', namespace='auth
or-polls', app_name='polls')),
    url(r'^publisher-polls/', include('polls.urls', namespace='p
ublisher-polls', app_name='polls')),
]
```

```
#polls/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='det
ail'),
    ...
]
```

根据以上设置，可以使用下面的查询：

- 如果其中一个实例是当前实例 —— 如果我们正在渲染' `author-polls` '实例的 `detail` 页面 —— ' `polls:index` '将解析成' `author-polls` '实例的主页面；例如下面两个都将解析成" `/author-polls/` "。

在基于类的视图的方法中：

```
reverse('polls:index', current_app=self.request.resolver_match.n
amespace)
```

和在模板中：

```
{% url 'polls:index' %}
```

注意，在模板中的反查需要添加`request`的`current_app`属性，像这样：

```
def render_to_response(self, context, **response_kwargs):
    self.request.current_app = self.request.resolver_match.names
pace
    return super(DetailView, self).render_to_response(context, *
*response_kwargs)
```

- 如果没有当前实例 —— 如果我们在站点的其它地方渲染一个页面 —— '`polls:index`'将解析到最后注册的`polls`的一个实例。因为没有默认的实例（命名空间为'polls'的实例），将使用注册的`polls`的最后一个实例。它将是'`publisher-polls`'，因为它是在`urlpatterns`中最后一个声明的。
- '`author-polls:index`'将永远解析到'`author-polls`'实例的主页（'`publisher-polls`'类似）。

如果还有一个默认的实例 —— 例如，一个名为'`polls`'的实例 —— 上面例子中唯一的变化是当没有当前实例的情况（上述第二种情况）。在这种情况下'`polls:index`'将解析到默认实例而不是`urlpatterns`中最后声明的实例的主页。

URL 命名空间和被包含的URLconf

被包含的URLconf 的命名空间可以通过两种方式指定。

首先，在你构造你的URL 模式时，你可以提供 应用 和 实例的命名空间给`include()`作为参数。例如：

```
url(r'^polls/', include('polls.urls', namespace='author-polls',
app_name='polls')),
```

这将包含`polls.urls`中定义的URL 到应用命名空间 '`polls`'中，其实例命名空间为'`author-polls`'。

其次，你可以include 一个包含嵌套命名空间数据的对象。如果你`include()`一个`url()`实例的列表，那么该对象中包含的URL 将添加到全局命名空间。然而，你还可以`include()`一个3个元素的元组：

```
(<list of url() instances>, <application namespace>, <instance n
amespace>)
```

例如：

```
from django.conf.urls import include, url

from . import views

polls_patterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='det
ail'),
]

url(r'^polls/', include((polls_patterns, 'polls', 'author-polls'
))),
```

这将include 命名的URL 模式到给定的应用和实例命名空间中。

例如，Django 的管理站点部署的实例叫 `AdminSite` 。 `AdminSite` 对象具有一个 `urls` 属性：一个3元组，包含管理站点中的所有URL 模式和应用的命名空间' `admin` '以及管理站点实例的名称。你 `include()` 到你项目的 `urlpatterns` 中的是这个 `urls` 属性。

请确保传递一个元组给 `include()` 。如果你只是传递3个参数： `include(polls_patterns, 'polls', 'author-polls')` ，Django 不会抛出一个错误，但是根据 `include()` 的功能，' `polls` '将是实例的命名空间而' `author-polls` '将是应用的命名空间，而不是反过来的。

> 译者：Django 文档协作翻译小组，原文：URLconfs。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写视图

一个视图函数，或者简短来说叫做视图，是一个简单的Python函数，它接受web请求，并且返回web响应。响应可以是一张网页的HTML内容，一个重定向，一个404错误，一个XML文档，或者一张图片. . . 是任何东西都可以。无论视图本身包含什么逻辑，都要返回响应。代码写在哪里也无所谓，只要它在你的Python目录下面。除此之外没有更多的要求了——可以说“没有什么神奇的地方”。为了能够把代码放在某个地方，惯例是把视图放在叫做views.py的文件中，然后把它放到你的项目或者应用目录里。

一个简单的视图

下面是一个返回当前日期和时间作为HTML文档的视图：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

让我们逐行阅读上面的代码：

- 首先，我们从 django.http模块导入了HttpResponse类，以及Python的datetime库。
- 接着，我们定义了current_datetime函数。它是一个视图函数。每个视图函数都应接收HttpRequest对象作为第一个参数，一般叫做request。
- 注意视图函数的名称并不重要；不需要用一个统一的命名方式来命名，以便让Django识别它。我们将其命名为current_datetime，是因为这个名称能够精确地反映出它的功能。
- 这个视图会返回一个HttpResponse对象，其中包含生成的响应。每个视图函数都要返回HttpResponse对象。（有例外，我们接下来会讲。）

> Django中的时区
>
> Django中包含一个TIME_ZONE设置，默认为America/Chicago。可能并不是你住的地方，所以你可能会在设置文件里修改它。

把你的**URL**映射到视图

所以，再重复一遍，这个视图函数返回了一个包含当前日期和时间的HTML页面。你需要创建URLconf来展示在特定的URL这一视图；详见URL 分发器。

返回错误

在Django中返回HTTP错误是相当容易的。有一些HttpResponse的子类代表不是200（“OK”）的HTTP状态码。你可以在request/response文档中找到所有可用的子类。你可以返回那些子类的一个实例，而不是普通的HttpResponse ，来表示一个错误。例如：

```
from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

由于一些状态码不太常用，所以不是每个状态码都有一个特化的子类。然而，如HttpResponse文档中所说的那样，你也可以向HttpResponse的构造器传递HTTP状态码，来创建你想要的任何状态码的返回类。例如：

```
from django.http import HttpResponse

def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

由于404错误是最常见的HTTP错误，所以处理这一错误的方式非常便利。

Http404异常

class django.http.Http404

当你返回一个像HttpResponseNotFound这样的错误时，它会输出这个错误页面的HTML作为结果：

```
return HttpResponseNotFound('<h1>Page not found</h1>')
```

为了便利起见，也因为你的站点有个一致的404页面是个好主意，Django提供了Http404异常。如果你在视图函数中的任何地方抛出Http404异常，Django都会捕获它，并且带上HTTP404错误码返回你应用的标准错误页面。

像这样：

```
from django.http import Http404
from django.shortcuts import render_to_response
from polls.models import Poll

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render_to_response('polls/detail.html', {'poll': p})
```

为了尽可能利用 Http404，你应该创建一个用来在404错误产生时展示的模板。这个模板应该叫做404.html，并且在你的模板树中位于最顶层。

如果你在抛出Http404异常时提供了一条消息，当DEBUG为True时它会出现在标准404模板的展示中。你可以将这些消息用于调试；但他们通常不适用于404模板本身。

自定义错误视图

Django中默认的错误视图对于大多数web应用已经足够了，但是如果你需要任何自定义行为，重写它很容易。只要在你的URLconf中指定下面的处理器（在其他任何地方设置它们不会有效）。

handler404覆盖了page_not_found()视图：

```
handler404 = 'mysite.views.my_custom_page_not_found_view'
```

handler500覆盖了server_error()视图：

```
handler500 = 'mysite.views.my_custom_error_view'
```

handler403覆盖了permission_denied()视图：

```
handler403 = 'mysite.views.my_custom_permission_denied_view'
```

handler400覆盖了bad_request()视图：

```
handler400 = 'mysite.views.my_custom_bad_request_view'
```

Django 的快捷函数

`django.shortcuts` 收集了“跨越” 多层MVC 的辅助函数和类。 换句话讲，这些函数/类为了方便，引入了可控的耦合。

render

```
render(request, template_name[, context][, context_instance][, conte
```

结合一个给定的模板和一个给定的上下文字典，并返回一个渲染后的 `HttpResponse` 对象。

`render()` 与以一个强制使用 `RequestContext` 的 `context_instance` 参数调用 `render_to_response()` 相同。

Django 不提供返回 `TemplateResponse` 的快捷函数，因为 `TemplateResponse` 的构造与 `render()` 提供的便利是一个层次的。

必选的参数

`request`

用于生成响应的请求对象。

`template_name`

要使用的模板的完整名称或者模板名称的一个序列。

可选的参数

`context`

添加到模板上下文的一个字典。默认是一个空字典。如果字典中的某个值是可调用的，视图将在渲染模板之前调用它。

```
Django 1.8 的改变：

context 参数之前叫做dictionary。这个名字在Django 1.8 中废弃并将在Djang
o 2.0 中删除。
```

`context_instance`

渲染模板的上下文实例。默认情况下，模板将使用 `RequestContext` 实例（ 值来自 `request` 和 `context` ）渲染。

版本 1.8 以后废弃：

废弃context_instance 参数。仅仅使用context。

`content_type`

生成的文档要使用的MIME 类型。默认为 `DEFAULT_CONTENT_TYPE` 设置的值。

`status`

响应的状态码。默认为200。

`current_app`

指示哪个应用包含当前的视图。更多信息，参见带命名空间的URL 的解析。

版本1.8 以后废弃：

废弃current_app 参数。你应该设置request.current_app。

`using`

用于加载模板使用的模板引擎的名称。

Changed in Django 1.8:

增加using 参数。

Changed in Django 1.7:

增加dirs 参数。

Deprecated since version 1.8:

废弃dirs 参数。

示例

下面的示例渲染模板 `myapp/index.html` ，MIME 类型为 `application/xhtml+xml` ：

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {"foo": "bar"},
        content_type="application/xhtml+xml")
```

这个示例等同于：

```
from django.http import HttpResponse
from django.template import RequestContext, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    c = RequestContext(request, {'foo': 'bar'})
    return HttpResponse(t.render(c),
        content_type="application/xhtml+xml")
```

render_to_response

`render_to_response(template_name[, context][, context_instance][, co`

根据一个给定的上下文字典渲染一个给定的目标，并返回渲染后的HttpResponse。

必选的参数

`template_name`

使用的模板的完整名称或者模板名称的序列。如果给出的是一个序列，将使用存在的第一个模板。关于如何查找模板的更多信息请参见 模板加载的文档 。

可选的参数

`context`

添加到模板上下文中的字典。默认是个空字典。如果字典中的某个值是可调用的，视图将在渲染模板之前调用它。

```
Changed in Django 1.8:

context 参数之前叫做dictionary。 这个名字在Django 1.8 中废弃并将在Djan
go 2.0 中删除。
```

`context_instance`

渲染模板使用的上下文实例。默认情况下，模板将 `Context` 实例（值来自 `context` ）渲染。如果你需要使用上下文处理器，请使用 `RequestContext` 实例渲染模板。你的代码看上去像是这样：

```
return render_to_response('my_template.html',
                          my_context,
                          context_instance=RequestContext(reques
t))
```

```
版本1.8 以后废弃：

废弃context_instance 参数。 仅仅使用context。
```

`content_type`

生成的文档使用的MIME 类型。默认为 `DEFAULT_CONTENT_TYPE` 设置的值。

`status`

相应的状态码。默认为200。

`using`

加载模板使用的模板引擎的名称。

```
Changed in Django 1.8:

添加status 和using 参数。
```

```
Changed in Django 1.7:

增加dirs 参数。
```

```
Deprecated since version 1.8:

废弃dirs 参数。
```

示例

下面的示例渲染模板 `myapp/index.html` ，MIIME 类型为 `application/xhtml+xml` ：

```
from django.shortcuts import render_to_response

def my_view(request):
    # View code here...
    return render_to_response('myapp/index.html', {"foo": "bar"}
,
        content_type="application/xhtml+xml")
```

这个示例等同于：

```
from django.http import HttpResponse
from django.template import Context, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    c = Context({'foo': 'bar'})
    return HttpResponse(t.render(c),
        content_type="application/xhtml+xml")
```

redirect

`redirect(to, [permanent=False, ]*args, **kwargs)[source]`

为传递进来的参数返回HttpResponseRedirect 给正确的URL 。

参数可以是：

- 一个模型：将调用模型的 `get_absolute_url()` 函数
- 一个视图，可以带有参数：将使用 `urlresolvers.reverse` 来反向解析名称
- 一个绝对的或相对的URL，将原样作为重定向的位置。

默认返回一个临时的重定向；传递 `permanent=True` 可以返回一个永久的重定向。

```
Django 1.7 中的改变：

增加使用相对URL 的功能。
```

示例

你可以用多种方式使用 `redirect()` 函数。

通过传递一个对象；将调用 `get_absolute_url()` 方法来获取重定向的URL：

```
from django.shortcuts import redirect

def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object)
```

通过传递一个视图的名称，可以带有位置参数和关键字参数；将使用 `reverse()` 方法反向解析URL：

```
def my_view(request):
    ...
    return redirect('some-view-name', foo='bar')
```

传递要重定向的一个硬编码的URL：

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

完整的URL 也可以：

```
def my_view(request):
    ...
    return redirect('http://example.com/')
```

默认情况下， `redirect()` 返回一个临时重定向。以上所有的形式都接收一个 `permanent` 参数；如果设置为 `True` ，将返回一个永久的重定向：

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object, permanent=True)
```

get_object_or_404

`get_object_or_404(klass, *args, **kwargs)[source]`

在一个给定的模型管理器上调用 `get()` ，但是引发 `Http404` 而不是模型的 `DoesNotExist` 异常。

必选的参数

`klass`

获取该对象的一个 `Model` 类，`Manager` 或 `QuerySet` 实例。

`**kwargs`

查询的参数，格式应该可以被 `get()` 和 `filter()` 接受。

示例

下面的示例从 `MyModel` 中使用主键1 来获取对象：

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)
```

这个示例等同于：

```
from django.http import Http404

def my_view(request):
    try:
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No MyModel matches the given query.")
```

最常见的用法是传递一个 `Model` ，如上所示。然而，你还可以传递一个 `QuerySet` 实例：

```
queryset = Book.objects.filter(title__startswith='M')
get_object_or_404(queryset, pk=1)
```

上面的示例有点做作，因为它等同于：

```
get_object_or_404(Book, title__startswith='M', pk=1)
```

但是如果你的queryset 来自其它地方，它就会很有用了。

最后你还可以使用一个管理器。如果你有一个自定义的管理器，它将很有用：

```
get_object_or_404(Book.dahl_objects, title='Matilda')
```

你还可以使用关联的 管理器：

```
author = Author.objects.get(name='Roald Dahl')
get_object_or_404(author.book_set, title='Matilda')
```

注：与get()一样，如果找到多个对象将引发一个 `MultipleObjectsReturned` 异常。

get_list_or_404

`get_list_or_404(klass, *args, **kwargs)[source]`

返回一个给定模型管理器上filter() 的结果，并将结果映射为一个列表，如果结果为空则返回Http404。

必选的参数

`klass`

获取该列表的一个 `Model` 、 `Manager` 或 `QuerySet` 实例。

`**kwargs`

查寻的参数，格式应该可以被 `get()` 和 `filter()` 接受。

示例

下面的示例从 `MyModel` 中获取所有发布出来的对象：

```
from django.shortcuts import get_list_or_404

def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

这个示例等同于：

```
from django.http import Http404

def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404("No MyModel matches the given query.")
```

> 译者：Django 文档协作翻译小组，原文：Shortcuts。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

视图装饰器

Django为视图提供了数个装饰器，用以支持相关的HTTP服务。

允许的HTTP 方法

`django.views.decorators.http` 包里的装饰器可以基于请求的方法来限制对视图的访问。若条件不满足会返回 `django.http.HttpResponseNotAllowed` 。

`require_http_methods` (request_method_list)[source]

限制视图只能服务规定的http方法。用法：

```
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it th
is far
    # ...
    pass
```

注意，方法名必须大写。

`require_GET()`

只允许视图接受GET方法的装饰器。

`require_POST()`

只允许视图接受POST方法的装饰器。

`require_safe()`

只允许视图接受 GET 和 HEAD 方法的装饰器。 这些方法通常被认为是安全的，因为方法不该有请求资源以外的目的。

> 注
>
> Django 会自动清除对HEAD 请求的响应中的内容而只保留头部，所以在你的视图中你处理HEAD 请求的方式可以完全与GET 请求一致。因为某些软件，例如链接检查器，依赖于HEAD 请求，所以你可能应该使用 `require_safe` 而不是 `require_GET` 。

可控制的视图处理

`django.views.decorators.http` 中的以下装饰器可以用来控制特定视图的缓存行为。

`condition` (etag_func=None, last_modified_func=None)[source]

`etag` (etag_func)[source]

`last_modified` (last_modified_func)[source]

这些装饰器可以用于生成ETag 和Last-Modified 头部；参考 conditional view processing.

GZip 压缩

`django.views.decorators.gzip` 里的装饰器基于每个视图控制其内容压缩。

`gzip_page()`

如果浏览器允许gzip 压缩，这个装饰器将对内容进行压缩。它设置相应的 `Vary` 头部，以使得缓存根据 `Accept-Encoding` 头来存储信息。

Vary 头部

`django.views.decorators.vary` 可以用来基于特定的请求头部来控制缓存。

`vary_on_cookie` (func)[source]

`vary_on_headers` (*headers)[source]

到当构建缓存的键时，Vary 头部定义一个缓存机制应该考虑的请求头。

参见使用vary 头部。

> 译者：Django 文档协作翻译小组，原文：Decorators。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

参考

内建的视图

有几个Django 的内建视图在编写视图 中讲述，文档的其它地方也会有所讲述。

开发环境中的文件服务器

static.serve(request, path, document_root, show_indexes=False)

在本地的开发环境中，除了你的项目中的静态文件，可能还有一些文件，出于方便，你希望让Django 来作为服务器。serve() 视图可以用来作为任意目录的服务器。（该视图不能用于生产环境，应该只用于开发时辅助使用；在生产环境中你应该使用一个真实的前端Web 服务器来服务这些文件）。

最常见的例子是用户上传文档到 `MEDIA_ROOT` 中。 `django.contrib.staticfiles` 用于静态文件且没有对用户上传的文件做处理，但是你可以通过在URLconf 中添加一些内容来让Django 作为MEDIA_ROOT 的服务器：

```
from django.conf import settings
from django.views.static import serve

# ... the rest of your URLconf goes here ...

if settings.DEBUG:
    urlpatterns += [
        url(r'^media/(?P<path>.*)$', serve, {
            'document_root': settings.MEDIA_ROOT,
        }),
   ]
```

注意，这里的代码片段假设你的 `MEDIA_URL` 的值为 `'/media/'` 。它将调用 `serve()` 视图，传递来自URLconf 的路径和（必选的） `document_root` 参数。

因为定义这个URL 模式显得有些笨拙，Django 提供一个小巧的URL 辅助函数 `static()` ，它接收 `MEDIA_URL` 这样的参数作为前缀和视图的路径如 `'django.views.static.serve'` 。其它任何函数参数都将透明地传递给视图。

错误视图

Django 原生自带几个默认视图用于处理HTTP 错误。若要使用你自定义的视图覆盖它们，请参见自定义错误视图。

404 (page not found) 视图

defaults.page_not_found(request, template_name='404.html')

当你在一个视图中引发Http404 时，Django 将加载一个专门的视图用于处理404 错误。默认为 `django.views.defaults.page_not_found()` 视图，它产生一个非常简单的“Not Found” 消息或者渲染 `404.html` 模板，如果你在根模板目录下创建了它的话。

默认的404 视图将传递一个变量给模板： `request_path` ，它是导致错误的URL。

关于404 视图需要注意的3点：

- 如果Django 在检测URLconf 中的每个正则表达式后没有找到匹配的内容也将调用404 视图。
- 404 视图会被传递一个 `RequestContext` 并且可以访问模板上下文处理器提供的变量（例如 `MEDIA_URL` ）。
- 如果DEBUG 设置为 `True` （在你的 `settings` 模块中），那么将永远不会调用404 视图，而是显示你的URLconf 并带有一些调试信息。

500 (server error) 视图

defaults.server_error(request, template_name='500.html')

类似地，在视图代码中出现运行时错误，Django 将执行特殊情况下的行为。如果一个视图导致异常，Django 默认情况下将调用 `django.views.defaults.server_error` 视图，它产生一个非常简单的“Server Error” 消息或者渲染 `500.html` ，如果你在你的根模板目录下定义了它的话。

默认的500 视图不会传递变量给 `500.html` 模板，且使用一个空 `Context` 来渲染以减少再次出现错误的可能性。

如果 `DEBUG` 设置为 `True` （在你的 `settings` 模块中），那么将永远不会调用500 视图，而是显示回溯并带有一些调试信息。

403 (HTTP Forbidden) 视图

defaults.permission_denied(request, template_name='403.html')

与404 和500 视图一样，Django 具有一个处理403 Forbidden 错误的视图。如果一个视图导致一个403 视图，那么Django 将默认调用 `django.views.defaults.permission_denied` 视图。

该视图加载并渲染你的根模板目录下的 `403.html` ，如果这个文件不存在则根据RFC 2616（HTTP 1.1 Specification）返回“403 Forbidden”文本。

`django.views.defaults.permission_denied` 通过 `PermissionDenied` 异常触发。若要拒绝访问一个视图，你可以这样视图代码：

```
from django.core.exceptions import PermissionDenied

def edit(request, pk):
    if not request.user.is_staff:
        raise PermissionDenied
    # ...
```

400 (bad request) 视图

defaults.bad_request(request, template_name='400.html')

当Django 中引发一个 `SuspiciousOperation` 时，它可能通过Django 的一个组件处理（例如重设会话的数据）。如果没有特殊处理，Django 将认为当前的请求时一个'bad request' 而不是一个server error。

`django.views.defaults.bad_request` 和server_error 视图非常相似，除了返回400 状态码来表示错误来自客户端的操作。

`bad_request` 视图同样只是在 `DEBUG` 为 `False` 时使用。

> 译者：Django 文档协作翻译小组，原文：Built-in Views。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Request 对象和Response 对象

概述

Django 使用Request 对象和Response 对象在系统间传递状态。

当请求一个页面时，Django会建立一个包含请求元数据的 `HttpRequest` 对象。当Django 加载对应的视图时， `HttpRequest` 对象将作为视图函数的第一个参数。每个视图会返回一个 `HttpResponse` 对象。

本文档对 `HttpRequest` 和 `HttpResponse` 对象的API 进行说明，这些API 定义在 `django.http` 模块中。

HttpRequest 对象

class `HttpRequest`

属性

下面除非特别说明，所有属性都认为是只读的。 `会话(session)` 属性是个例外，需要注意。

`HttpRequest.scheme`

New in Django 1.7.

一个字符串，表示请求的方案（通常是 `http` 或 `https` ）。

`HttpRequest.body`

一个字节字符串，表示原始HTTP 请求的正文。它对于处理非HTML 形式的数据非常有用：二进制图像、XML等。 如果要处理常规的表单数据，应该使用 `HttpRequest.POST` 。

你也可以使用"类文件"形式的接口从HttpRequest 中读取数据。参见 `HttpRequest.read()` 。

`HttpRequest.path`

一个字符串，表示请求的页面的完整路径，不包含域名。

例如： `"/music/bands/the_beatles/"`

`HttpRequest.path_info`

在某些Web 服务器配置下，主机名后的URL 部分被分成脚本前缀部分和路径信息部分。 `path_info` 属性将始终包含路径信息部分，不论使用的Web 服务器是什么。使用它代替 `path` 可以让代码在测试和开发环境中更容易地切换。

例如，如果应用的 `WSGIScriptAlias` 设置为 `"/minfo"` ，那么当 `path` 是 `"/minfo/music/bands/the_beatles/"` 时 `path_info` 将是 `"/music/bands/the_beatles/"` 。

`HttpRequest.method`

一个字符串，表示请求使用的HTTP 方法。必须使用大写。例如：

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

`HttpRequest.encoding`

一个字符串，表示提交的数据的编码方式（如果为 `None` 则表示使用 `DEFAULT_CHARSET` 设置）。这个属性是可写的，你可以修改它来修改访问表单数据使用的编码。接下来对属性的任何访问（例如从 `GET` 或 `POST` 中读取数据）将使用新的 `encoding` 值。如果你知道表单数据的编码不在 `DEFAULT_CHARSET` 中，则使用它。

`HttpRequest.GET`

一个类似于字典的对象，包含HTTP GET 的所有参数。详情请参考下面的 `QueryDict` 文档。

`HttpRequest.POST`

一个包含所有给定的HTTP POST参数的类字典对象，提供了包含表单数据的请求。详情请参考下面的 `QueryDict` 文档。如果需要访问请求中的原始或非表单数据，可以使用 `HttpRequest.body` 属性。

POST 请求可以带有空的 `POST` 字典 —— 如果通过HTTP POST 方法请求一个表单但是没有包含表单数据的话。因此，不应该使用 `if request.POST` 来检查使用的是否是POST 方法；应该使用 `if request.method == "POST"` （参见上文）。

注意： `POST` 不包含上传的文件信息。参见 `FILES` 。

`HttpRequest.REQUEST`

Deprecated since version 1.7: 使用更显式的 `GET` 和 `POST` 代替。

一个类似于字典的对象，它首先搜索 `POST` ，然后搜索 `GET` ，主要是为了方便。灵感来自于PHP 的 `$_REQUEST` 。

例如，如果 `GET = {"name": "john"}` 而 `POST = {"age": '34'}` ， `REQUEST["name"]` 将等于 `"john"` 以及 `REQUEST["age"]` 将等于 `"34"` 。

强烈建议使用 `GET` 和 `POST` 而不要用 `REQUEST` ，因为它们更加明确。

`HttpRequest.COOKIES`

一个标准的Python 字典，包含所有的cookie。键和值都为字符串。

`HttpRequest.FILES`

一个类似于字典的对象，包含所有的上传文件。 `FILES` 中的每个键为 `<input type="file" name="" />` 中的 `name` 。

更多信息参见管理文件。

注意， `FILES` 只有在请求的方法为POST 且提交的 `<form>` 带有 `enctype="multipart/form-data"` 的情况下才会包含数据。否则， `FILES` 将为一个空的类似于字典的对象。

`HttpRequest.META`

一个标准的Python 字典，包含所有的HTTP 头部。具体的头部信息取决于客户端和服务器，下面是一些示例：

- `CONTENT_LENGTH` —— 请求的正文的长度（是一个字符串）。
- `CONTENT_TYPE` —— 请求的正文的MIME 类型。
- `HTTP_ACCEPT` —— 响应可接收的Content-Type。
- `HTTP_ACCEPT_ENCODING` —— 响应可接收的编码。
- `HTTP_ACCEPT_LANGUAGE` ?—— 响应可接收的语言。
- `HTTP_HOST` —— 客服端发送的HTTP Host 头部。
- `HTTP_REFERER` —— Referring 页面。
- `HTTP_USER_AGENT` —— 客户端的user-agent 字符串。
- `QUERY_STRING` —— 单个字符串形式的查询字符串（未解析过的形式）。
- `REMOTE_ADDR` —— 客户端的IP 地址。
- `REMOTE_HOST` —— 客户端的主机名。
- `REMOTE_USER` —— 服务器认证后的用户。
- `REQUEST_METHOD` —— 一个字符串，例如 `"GET"` 或 `"POST"` 。
- `SERVER_NAME` —— 服务器的主机名。
- `SERVER_PORT` —— 服务器的端口（是一个字符串）。

从上面可以看到，除 `CONTENT_LENGTH` 和 `CONTENT_TYPE` 之外，请求中的任何HTTP 头部转换为 `META` 的键时，都会将所有字母大写并将连接符替换为下划线最后加上 `HTTP_` 前缀。所以，一个叫做 `X-Bender` 的头部将转换成 `META` 中的 `HTTP_X_BENDER` 键。

`HttpRequest.user`

一个 `AUTH_USER_MODEL` 类型的对象，表示当前登录的用户。如果用户当前没有登录， `user` 将设置为 `django.contrib.auth.models.AnonymousUser` 的一个实例。你可以通过 `is_authenticated()` 区分它们，像这样：

```
if request.user.is_authenticated():
    # Do something for logged-in users.
else:
    # Do something for anonymous users.
```

`user` 只有当Django 启用 `AuthenticationMiddleware` 中间件时才可用。更多信息，参见*Django 中的用户认证*。

`HttpRequest.session`

一个既可读又可写的类似于字典的对象，表示当前的会话。只有当Django 启用会话的支持时才可用。完整的细节参见会话的文档。

`HttpRequest.urlconf`

不是由Django 自身定义的，但是如果其它代码（例如，自定义的中间件类）设置了它，Django 就会读取它。如果存在，它将用来作为当前的请求的Root URLconf，并覆盖 `ROOT_URLCONF` 设置。细节参见*Django 如何处理请求*。

`HttpRequest.resolver_match`

一个 `ResolverMatch` 的实例，表示解析后的URL。这个属性只有在URL 解析方法之后才设置，这意味着它在所有的视图中可以访问，但是在在URL 解析发生之前执行的中间件方法中不可以访问（比如 `process_request` ，但你可以使用 `process_view` 代替）。

方法

`HttpRequest.get_host` ()

根据从 `HTTP_X_FORWARDED_HOST` （如果打开 `USE_X_FORWARDED_HOST` ）和 `HTTP_HOST` 头部信息返回请求的原始主机。如果这两个头部没有提供相应的值，则使用 `SERVER_NAME` 和 `SERVER_PORT` ，在**PEP 3333** 中有详细描述。

例如： `"127.0.0.1:8000"`

注

当主机位于多个代理的后面， `get_host()` 方法将会失败。有一个解决办法是使用中间件重写代理的头部，例如下面的例子：

```
class MultipleProxyMiddleware(object):
    FORWARDED_FOR_FIELDS = [
        'HTTP_X_FORWARDED_FOR',
        'HTTP_X_FORWARDED_HOST',
        'HTTP_X_FORWARDED_SERVER',
    ]

    def process_request(self, request):
        """
 Rewrites the proxy headers so that only the most
 recent proxy is used.
 """
        for field in self.FORWARDED_FOR_FIELDS:
            if field in request.META:
                if ',' in request.META[field]:
                    parts = request.META[field].split(',')
                    request.META[field] = parts[-1].strip()
```

这个中间件应该放置在所有依赖于 `get_host()` 的中间件之前 —— 例如， `CommonMiddleware` 和 `CsrfViewMiddleware` 。

`HttpRequest.get_full_path` ()

返回 `path` ，如果可以将加上查询字符串。

例如： `"/music/bands/the_beatles/?print=true"`

`HttpRequest.build_absolute_uri` (*location*)

返回 `location` 的绝对URI。如果location 没有提供，则设置为 `request.get_full_path()` 。

如果URI 已经是一个绝对的URI，将不会修改。否则，使用请求中的服务器相关的变量构建绝对URI。

例如： `"http://example.com/music/bands/the_beatles/?print=true"`

`HttpRequest.get_signed_cookie` (*key*, *default=RAISE_ERROR*, *salt=''*, *max_age=None*)

返回签名过的Cookie 对应的值，如果签名不再合法则返回 `django.core.signing.BadSignature` 。如果提供 `default` 参数，将不会引发异常并返回default 的值。

可选参数 `salt` 可以用来对安全密钥强力攻击提供额外的保护。 `max_age` 参数用于检查Cookie 对应的时间戳以确保Cookie 的时间不会超过 `max_age` 秒。

示例：

```
>>> request.get_signed_cookie('name')
'Tony'
>>> request.get_signed_cookie('name', salt='name-salt')
'Tony' # assuming cookie was set using the same salt
>>> request.get_signed_cookie('non-existing-cookie')
...
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name', max_age=60)
...
SignatureExpired: Signature age 1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name', False, max_age=60)
False
```

更多信息参见密钥签名。

`HttpRequest.is_secure` ()

如果请求时是安全的，则返回 `True` ；即请求是通过HTTPS 发起的。

`HttpRequest.is_ajax` ()

如果请求是通过 `XMLHttpRequest` 发起的，则返回 `True` ，方法是检查 `HTTP_X_REQUESTED_WITH` 头部是否是字符串 `'XMLHttpRequest'` 。大部分现代的JavaScript 库都会发送这个头部。如果你编写自己的XMLHttpRequest 调用（在浏览器端），你必须手工设置这个值来让 `is_ajax()` 可以工作。

如果一个响应需要根据请求是否是通过AJAX 发起的，并且你正在使用某种形式的缓存例如Django 的 `cache middleware` ， 你应该使用 `vary_on_headers('HTTP_X_REQUESTED_WITH')` 装饰你的视图以让响应能够正确地缓存。

`HttpRequest.read` (*size=None*)

`HttpRequest.readline` ()

`HttpRequest.readlines` ()

`HttpRequest.xreadlines` ()

`HttpRequest.__iter__` ()

这几个方法实现类文件的接口用于读取HttpRequest· 实例。这使得可以用流的方式读取进来的请求。常见的使用常见是使用迭代的解析器处理一个大型的XML而不用在内存中构建一个完整的XML 树。

根据这个标准的接口，一个HttpRequest 实例可以直接传递给XML 解析器，例如ElementTree：

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

QueryDict 对象

class `QueryDict`

在 `HttpRequest` 对象中， `GET` 和 `POST` 属性是`django.http.QueryDict` 的实例，它是一个自定义的类似字典的类，用来处理同一个键带有多个值。这个类的需求来自某些HTML 表单元素传递多个值给同一个键， `<select multiple>` 是一个显著的例子。

`request.POST` 和 `request.GET` 的 `QueryDict` 在一个正常的请求/响应循环中是不可变的。若要获得可变的版本，需要使用 `.copy()` 。

方法

`QueryDict` 实现了字典的所有标准方法，因为它是字典的子类。下面列出了不同点：

`QueryDict.__init__` (*query_string=None*, *mutable=False*, *encoding=None*)

基于 `query_string` 实例化 `QueryDict` 一个对象。

```
>>> QueryDict('a=1&a=2&c=3')
<QueryDict: {'a': ['1', '2'], 'c': ['3']}>
```

If `query_string` 没被传入, ? `QueryDict` 的结果 是空的 （将没有键和值).

你所遇到的大部分?对象都是不可修改的，例如request.POST 和 `request.GET`。 如果需要实例化你自己的可以修改的对象，通过往它的 `__init__()` 方法来传递参数 `mutable=True` 可以实现。 .

设置键和值的字符串都将从 `encoding` 转换为unicode。如果没有指定编码的话，默认会设置为 `DEFAULT_CHARSET` .

Changed in Django 1.8:

在以前的版本中， `query_string` 是一个必需的位置参数。

`QueryDict.__getitem__` (*key*)

返回给出的key 的值。如果key 具有多个值， `__getitem__()` 返回最新的值。如果key 不存在，则引发 `django.utils.datastructures.MultiValueDictKeyError` 。（它是Python 标准 `KeyError` 的一个子类，所以你仍然可以坚持捕获 `KeyError` 。）

`QueryDict.__setitem__` (*key*, *value*)

设置给出的key 的值为 `[value]` （一个Python 列表，它具有唯一一个元素 `value` ）。注意，这和其它具有副作用的字典函数一样，只能在可变的 `QueryDict` 上调用（如通过 `copy()` 创建的字典）。

`QueryDict.__contains__` (*key*)

如果给出的key 已经设置，则返回 `True` 。它让你可以做 `if "foo" in request.GET` 这样的操作。

`QueryDict.get` (*key*, *default*)

使用与上面 `__getitem__()` 相同的逻辑，但是当key 不存在时返回一个默认值。

`QueryDict.setdefault` (*key*, *default*)

类似标准字典的 `setdefault()` 方法，只是它在内部使用的是 `__setitem__()` 。

`QueryDict.update` (*other_dict*)

接收一个 `QueryDict` 或标准字典。类似标准字典的 `update()` 方法，但是它附加到当前字典项的后面，而不是替换掉它们。例如：

```
>>> q = QueryDict('a=1', mutable=True)
>>> q.update({'a': '2'})
>>> q.getlist('a')
['1', '2']
>>> q['a'] # returns the last
['2']
```

`QueryDict.items` ()

类似标准字典的 `items()` 方法，但是它使用的是和 `__getitem__` 一样返回最新的值的逻辑。例如：

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[('a', '3')]
```

`QueryDict.iteritems` ()

类似标准字典的 `iteritems()` 方法。类似 `QueryDict.items()` ，它使用的是和 `QueryDict.__getitem__()` 一样的返回最新的值的逻辑。

`QueryDict.iterlists` ()

类似 `QueryDict.iteritems()` ，只是它将字典中的每个成员作为列表。

`QueryDict.values` ()

类似标准字典的 `values()` 方法，但是它使用的是和 `__getitem__` 一样返回最新的值的逻辑。例如：

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
['3']
```

`QueryDict.itervalues` ()

类似 `QueryDict.values()` ，只是它是一个迭代器。

另外， `QueryDict` 具有以下方法：

`QueryDict.copy` ()

返回对象的副本，使用Python 标准库中的 `copy.deepcopy()` 。此副本是可变的即使原始对象是不可变的。

`QueryDict.getlist` (*key*, *default*)

以Python 列表形式返回所请求的键的数据。如果键不存在并且没有提供默认值，则返回空列表。它保证返回的是某种类型的列表，除非默认值不是列表。

`QueryDict.setlist` (*key*, *list_*)

设置给定的键为 `list_` （与 `__setitem__()` 不同)。

`QueryDict.appendlist` (*key*, *item*)

将项追加到内部与键相关联的列表中。

`QueryDict.setlistdefault` (*key*, *default_list*)

类似 `setdefault` ，除了它接受一个列表而不是单个值。

`QueryDict.lists` ()

类似 `items` ，只是它将字典中的每个成员作为列表。例如：

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[('a', ['1', '2', '3'])]
```

`QueryDict.pop` (*key*)

返回给定键的值的列表，并从字典中移除它们。如果键不存在，将引发 `KeyError` 。例如 ：

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.pop('a')
['1', '2', '3']
```

`QueryDict.popitem` ()

删除字典任意一个成员（因为没有顺序的概念），并返回二值元组，包含键和键的所有值的列表。在一个空的字典上调用时将引发 `KeyError` 。例如 ：

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.popitem()
('a', ['1', '2', '3'])
```

`QueryDict.dict` ()

返回 `QueryDict` 的 `dict` 表示形式。对于 `QueryDict` 中的每个(key, list)对 ， `dict` 将有(key, item) 对，其中item 是列表中的一个元素，使用与 `QueryDict.__getitem__()` 相同的逻辑：

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
{'a': '5'}
```

`QueryDict.urlencode` ([*safe*])

在查询字符串格式返回数据的字符串形式。示例：

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

可选地，urlencode 可以传递不需要编码的字符。例如：

```
>>> q = QueryDict(mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```

HttpResponse 对象

class `HttpResponse`

与由Django自动创建的 `HttpRequest` 对象相比， `HttpResponse` 对象由程序员创建.你创建的每个视图负责初始化实例,填充并返回一个 `HttpResponse` .

? `HttpResponse` 类是在 `django.http` 模块中定义的。

用法

传递字符串

典型的应用是传递一个字符串作为页面的内容到 `HttpResponse` 构造函数:

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="
text/plain")
```

如果你想增量增加内容，你可以将 `response` 看做一个类文件对象

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

传递迭代器

最后你可以传递给 `HttpResponse` 一个迭代器而不是字符串. `HttpResponse` 将立即处理这个迭代器, 把它的内容存成字符串，并丢弃它

如果你需要从迭代器到客户端的数据数据流的形式响应, 你必须用 `StreamingHttpResponse` 类代替;.

配置 header fields

把它当作一个类似字典的结构，从你的response中设置和移除一个header field。

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

注意！与字典不同的是，如果要删除的header field不存在， `del` 不会抛出 `KeyError` 异常。

For setting the `Cache-Control` and `Vary` header fields, it is recommended to use the `patch_cache_control()` and `patch_vary_headers()` methods from `django.utils.cache`, since these fields can have multiple, comma-separated values. “补丁”方法确保其他值，例如，通过中间件添加的，不会删除。

HTTP header fields 不能包含换行。当我们尝试让header field包含一个换行符（CR 或者 LF），那么将会抛出一个 `BadHeaderError` 异常。

告诉浏览器以文件附件的形式处理服务器的响应

让浏览器以文件附件的形式处理响应, 需要声明 `content_type` 类型 和设置 `Content-Disposition` 头信息. 例如，下面是 如何给浏览器返回一个微软电子表格：

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename="foo.xls"'
```

There’s nothing Django-specific about the `Content-Disposition` header, but it’s easy to forget the syntax, so we’ve included it here.

属性

`HttpResponse.content`

一个用来代替content的字节字符串，如果必要，则从一个Unicode对象编码而来。

`HttpResponse.charset`

New in Django 1.8.

一个字符串，用来表示response将会被编码的字符集。如果没有在 `HttpResponse` 实例化的时候给定这个字符集，那么将会从 `content_type` 中解析出来。并且当这种解析成功的时候， `DEFAULT_CHARSET` 选项将会被使用。

`HttpResponse.status_code`

响应（response）的HTTP 响应状态码

`HttpResponse.reason_phrase`

The HTTP reason phrase for the response.

`HttpResponse.streaming`

这个选项总是 `False` 。

由于这个属性的存在，使得中间件（middleware）能够却别对待流式response和常规response。

`HttpResponse.closed`

New in Django 1.8.

`True` if the response has been closed.

方法

`HttpResponse.__init__` (*content=''*, *content_type=None*, *status=200*, *reason=None*, *charset=None*)

使用页面的内容（content）和content-type来实例化一个 `HttpResponse` 对象。

`content` 应该是一个迭代器或者字符串。如果它是一个迭代器，那么他应该返回的是一串字符串，并且这些字符串连接起来形成response的内容（content）。如果不是迭代器或者字符串，那么在其被接收的时候将转换成字符串。

`content_type` 是可选地通过字符集编码完成的MIME类型，并且用于填充HTTP `Content-Type` 头部。如果没有设定，会从 `DEFAULT_CONTENT_TYPE` 和 `DEFAULT_CHARSET` 设定中提取，作为默认值："*text/html;* charset=utf-8"。

`status` 是 HTTP 响应状态码 。.

`reason` 是HTTP响应短语 如果没有指定，则使用默认响应短语.

`charset` 在response中被编码的字符集。如果没有给定，将会从 `content_type` 中提取，如果提取不成功，那么 `DEFAULT_CHARSET` 的设定将被使用.

New in Django 1.8:

The `charset` parameter was added.

`HttpResponse.__setitem__` (*header*, *value*)

由给定的首部名称和值设定相应的报文首部。 `header` 和 `value` 都应该是字符串类型。

`HttpResponse.__delitem__` (*header*)

根据给定的首部名称来删除报文中的首部。如果对应的首部不存在将沉默地（不引发异常）失败。不区分大小写。

`HttpResponse.__getitem__` (*header*)

根据首部名称返回其值。不区分大小写。

`HttpResponse.has_header` (*header*)

通过检查首部中是否有给定的首部名称（不区分大小写），来返回 `True` 或 `False` 。

`HttpResponse.setdefault` (*header*, *value*)

New in Django 1.8.

设置一个首部，除非该首部 header 已经存在了。

`HttpResponse.set_cookie` (*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=False*)

设置一个Cookie。参数与Python 标准库中的 `Morsel` Cookie 对象相同。

- `max_age` 以秒为单位，如果Cookie 只应该持续客户端浏览器的会话时长则应该为 `None` （默认值）。如果没有指定 `expires` ，则会通过计算得到。
- `expires` 应该是一个?UTC `"Wdy, DD-Mon-YY HH:MM:SS GMT"` 格式的字符串，或者一个 `datetime.datetime` 对象。如果 `expires` 是一个 `datetime` 对象，则 `max_age` 会通过计算得到。
- 如果你想设置一个跨域的Cookie，请使用 `domain` 参数。例如， `domain=".lawrence.com"` 将设置一个www.lawrence.com、blogs.lawrence.com 和calendars.lawrence.com 都可读的Cookie。否则，Cookie 将只能被设置它的域读取。
- 如果你想阻止客服端的JavaScript 访问Cookie，可以设置 `httponly=True` 。

  HTTPOnly 是包含在HTTP 响应头部Set-Cookie 中的一个标记。它不是**RFC 2109** 中Cookie 的标准，也并没有被所有的浏览器支持。但是，如果使用，它是一种降低客户端脚本访问受保护的Cookie 数据风险的有用的方法。

警告

RFC 2109 和**RFC 6265** 都声明客户端至少应该支持4096 个字节的Cookie。对于许多浏览器，这也是最大的大小。如果视图存储大于4096 个字节的Cookie，Django 不会引发异常，但是浏览器将不能正确设置Cookie。

`HttpResponse.set_signed_cookie` (*key*, *value*, *salt=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=True*)

与 `set_cookie()` 类似，但是在设置之前将用密钥签名。通常与 `HttpRequest.get_signed_cookie()` 一起使用。你可以使用可选的 `salt` 参考来增加密钥强度，但需要记住将它传递给对应的 `HttpRequest.get_signed_cookie()` 调用。

`HttpResponse.delete_cookie` (*key*, *path='/'*, *domain=None*)

删除指定的key 的Cookie。如果key 不存在则什么也不发生。

由于Cookie 的工作方式， `path` 和 `domain` 应该与 `set_cookie()` 中使用的值相同 —— 否则Cookie 不会删掉。

`HttpResponse.write` (*content*)

此方法使 `HttpResponse` 实例是一个类似文件的对象。

`HttpResponse.flush` ()

此方法使 `HttpResponse` 实例是一个类似文件的对象。

`HttpResponse.tell` ()

此方法使 `HttpResponse` 实例是一个类似文件的对象。

`HttpResponse.getvalue` ()

New in Django 1.8.

返回 `HttpResponse.content` 的值。此方法使 `HttpResponse` 实例是一个类似流的对象。

`HttpResponse.writable` ()

New in Django 1.8.

始终为 `True` 。此方法使 `HttpResponse` 实例是一个类似流的对象。

`HttpResponse.writelines` (*lines*)

New in Django 1.8.

将一个包含行的列表写入响应。不添加行分隔符。此方法使 `HttpResponse` 实例是一个类似流的对象。

HttpResponse的子类

Django包含了一系列的 `HttpResponse` 衍生类（子类），用来处理不同类型的HTTP 响应（response）。与 `HttpResponse` 相同, 这些衍生类（子类）存在于 `django.http` 之中。

class `HttpResponseRedirect`

构造函数的第一个参数是必要的 — 用来重定向的地址。这些能够是完全特定的URL地址（比如， `'http://www.yahoo.com/search/'` ），或者是一个不包含域名的绝对路径地址（例如，? `'/search/'` ）。关于构造函数的其他参数，可以参见? `HttpResponse` 。注意！这个响应会返回一个302的HTTP状态码。

`url`

这个只读属性，代表响应将会重定向的URL地址（相当于 `Location` response hader）。

class `HttpResponsePermanentRedirect`

与 `HttpResponseRedirect` 一样，但是它会返回一个永久的重定向（HTTP状态码301）而不是一个“found”重定向（状态码302）。

class `HttpResponseNotModified`

构造函数不会有任何的参数，并且不应该向这个响应（response）中加入内容（content）。使用此选项可指定自用户上次请求（状态代码304）以来尚未修改页面。

class `HttpResponseBadRequest`

与 `HttpResponse` 的行为类似，但是使用了一个400的状态码。

class `HttpResponseNotFound`

与 `HttpResponse` 的行为类似，但是使用的404状态码。

class `HttpResponseForbidden`

与 `HttpResponse` 类似，但使用403状态代码。

class `HttpResponseNotAllowed`

与 `HttpResponse` 类似，但使用405状态码。构造函数的第一个参数是必须的：一个允许使用的方法构成的列表（例如， `['GET', 'POST']` ）。

class `HttpResponseGone`

与 `HttpResponse` 类似，但使用410状态码。

class `HttpResponseServerError`

与 `HttpResponse` 类似，但使用500状态代码。

注意

如果 `HttpResponse` 的自定义子类实现了 `render` 方法，Django会将其视为模拟 `SimpleTemplateResponse` ，且 `render` 方法必须自己返回一个有效的响应对象。

JsonResponse 对象

New in Django 1.7.

class `JsonResponse`

`JsonResponse.__init__` (*data*, *encoder=DjangoJSONEncoder*, *safe=True*, ***kwargs*)

`HttpResponse` 的一个子类，用户帮助创建JSON 编码的响应。它从父类继承大部分行为，并具有以下不同点：

它的默认 `Content-Type` 头部设置为 `application/json` 。

它的第一个参数 `data` ，应该为一个 `dict` 实例。如果 `safe` 参数设置为 `False` ，它可以是任何可JSON 序列化的对象。

`encoder` ，默认为 `django.core.serializers.json.DjangoJSONEncoder` ，用于序列化data。关于这个序列化的更多信息参见*JSON 序列化*。

布尔参数 `safe` 默认为 `True` 。如果设置为 `False` ，可以传递任何对象进行序列化（否则，只允许 `dict` 实例）。如果 `safe` 为 `True` ，而第一个参数传递的不是 `dict` 对象，将抛出一个 `TypeError` 。

用法

典型的用法如下：

```
>>> from django.http import JsonResponse
>>> response = JsonResponse({'foo': 'bar'})
>>> response.content
'{"foo": "bar"}'
```

序列化非字典对象

若要序列化非 `dict` 对象，你必须设置 `safe` 参数为 `False` ：

```
>>> response = JsonResponse([1, 2, 3], safe=False)
```

如果不传递 `safe=False` ，将抛出一个 `TypeError` 。

警告

在EcmaScript 第5版之前，这可能会使JavaScript `Array` 构造函数崩溃。出于这个原因，Django 默认不允许传递非字典对象给 `JsonResponse` 构造函数。然而，现代的大部分浏览器都已经实现EcmaScript 5，它删除了这种攻击性的数组。所以可以不用关注这个安全预防措施。

修改默认的JSON 编码器

如果你需要使用不同的JSON 编码器类，你可以传递 `encoder` 参数给构造函数：

```
>>> response = JsonResponse(data, encoder=MyJSONEncoder)
```

StreamingHttpResponse objects

class `StreamingHttpResponse`

`StreamingHttpResponse` 类被用来从Django流式化一个响应（response）到浏览器。如果生成响应太长或者是有使用太多的内存，你可能会想要这样做。例如，它对于生成大型*CSV*文件非常有用。

基于性能的考虑

Django是为了那些短期的请求（request）设计的。流式响应将会为整个响应期协同工作进程。这可能导致性能变差。

总的来说，你需要将代价高的任务移除请求—响应的循环，而不是求助于流式响应。

`StreamingHttpResponse` 不是 `HttpResponse` 的衍生类（子类），因为它实现了完全不同的应用程序接口（API）。尽管如此，除了以下的几个明显不同的地方，其他几乎完全相同：

- 应该提供一个迭代器给它，这个迭代器生成字符串来构成内容（content）
- 你不能直接访问它的内容（content），除非迭代它自己的响应对象。这只在响应被返回到客户端的时候发生。
- 它没有 `content` 属性。取而代之的是，它有一个 `streaming_content` 属性。
- 你不能使用类似文件对象的 `tell()` 或者 `write()` 方法。那么做会抛出一个异常。

`StreamingHttpResponse` should only be used in situations where it is absolutely required that the whole content isn't iterated before transferring the data to the client. 由于内容无法访问，许多中间件无法正常工作。例如，无法为流式响应生成 `ETag` 和 `Content-Length` 头。

属性

`StreamingHttpResponse.streaming_content`

一个表示内容（content）的字符串的迭代器

`StreamingHttpResponse.status_code`

响应的HTTP状态码。

`StreamingHttpResponse.reason_phrase`

响应的HTTP原因短语。

`StreamingHttpResponse.streaming`

这个选项总是 `True` .

FileResponse 对象

New in Django 1.8.

class `FileResponse`

`FileResponse` 是 `StreamingHttpResponse` 的衍生类（子类），为二进制文件做了优化。如果?wsgi server 来提供，则使用了 wsgi.file_wrapper ，否则将会流式化一个文件为一些小块。

`FileResponse` 需要通过二进制模式打开文件，如下:

```
>>> from django.http import FileResponse
>>> response = FileResponse(open('myfile.png', 'rb'))
```

TemplateResponse 和 SimpleTemplateResponse

标准的 `HttpResponse` 对象是静态的结构。在构造的时候提供给它们一个渲染之前的内容，但是当内容改变时它们却不能很容易地完成相应的改变。

然而，有时候允许装饰器或者中间件在响应被构造之后修改它是很有用的。例如，你可能想改变使用的模板，或者添加额外的数据到上下文中。

`TemplateResponse` 提供了实现这一点的方法。与基本的HttpResponse 对象不同， `TemplateResponse` 对象会记住视图提供的模板和上下文的详细信息来计算响应。响应的最终结果在后来的响应处理过程中直到需要时才计算。

SimpleTemplateResponse 对象

`class SimpleTemplateResponse[source]`

属性

`SimpleTemplateResponse.template_name`

渲染的模板的名称。接收一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

例如： `['foo.html', 'path/to/bar.html']`

> Deprecated since version 1.8:
>
> template_name 以前只接受一个 Template。

`SimpleTemplateResponse.context_data`

渲染模板时用到的上下文数据。它必须是一个 `dict` 。

例如： `{'foo': 123}`

> Deprecated since version 1.8:
>
> context_data 以前只接受一个 Context。

`SimpleTemplateResponse.rendered_content[source]`

使用当前的模板和上下文数据渲染出来的响应内容。

`SimpleTemplateResponse.is_rendered[source]`

一个布尔值，表示响应内容是否已经渲染。

方法

`SimpleTemplateResponse.__init__(template, context=None, content_type`

使用给定的模板、上下文、Content-Type、HTTP 状态和字符集初始化一个 `SimpleTemplateResponse` 对象。

`template`

一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

```
Deprecated since version 1.8:

template 以前只接受一个Template。
```

`context`

一个 `dict` ，包含要添加到模板上下文中的值。它默认是一个空的字典。

```
Deprecated since version 1.8:

context 以前只接受一个Context。
```

`content_type`

HTTP `Content-Type` 头部包含的值，包含MIME 类型和字符集的编码。 如果指定 `content_type` ，则使用它的值。 否则，使用 `DEFAULT_CONTENT_TYPE` 。

`status`

响应的HTTP 状态码。

`charset`

响应编码使用的字符集。 如果没有给出则从 `content_type` 中提取，如果提取不成功则使用 `DEFAULT_CHARSET` 设置。

`using`

加载模板使用的模板引擎的名称。

```
Changed in Django 1.8:

添加charset 和using 参数。
```

`SimpleTemplateResponse.resolve_context(context)[source]`

预处理即将用于渲染模板的上下文数据。接受包含上下文数据的一个 `dict` 。默认返回同一个 `dict` 。

若要自定义上下文，请覆盖这个方法。

Changed in Django 1.8:

resolve_context 返回一个dict。它以前返回一个Context。

Deprecated since version 1.8:

resolve_context 不再接受Context。

`SimpleTemplateResponse.resolve_template(template)[source]`

解析渲染用到的模板实例。接收一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

返回将被渲染的模板对象。

若要自定义模板的加载，请覆盖这个方法。

Changed in Django 1.8:

resolve_template 返回一个与后端有关的模板对象。它以前返回一个Template。

Deprecated since version 1.8:

resolve_template 不再接受一个 Template。

`SimpleTemplateResponse.add_post_render_callback()[source]`

添加一个渲染之后调用的回调函数。这个钩子可以用来延迟某些特定的处理操作（例如缓存）到渲染之后。

如果 `SimpleTemplateResponse` 已经渲染，那么回调函数将立即执行。

调用的时只传递给回调函数一个参数 —— 渲染后的 `SimpleTemplateResponse` 实例。

如果回调函数返回非 `None` 值，它将用作响应并替换原始的响应对象（以及传递给下一个渲染之后的回调函数，以此类推）。

`SimpleTemplateResponse.render()[source]`

设置 `response.content` 为 `SimpleTemplateResponse.rendered_content` 的结果，执行所有渲染之后的回调函数，最后返回生成的响应对象。

`render()` 只在第一次调用它时其作用。以后的调用将返回第一次调用的结果。

TemplateResponse 对象

`class TemplateResponse[source]`

`TemplateResponse` 是 `SimpleTemplateResponse` 的子类，而且能知道当前的 `HttpRequest` 。

方法

`TemplateResponse.__init__(request, template, context=None, content_t`

使用给定的模板、上下文、 `Content-Type` 、HTTP 状态和字符集实例化一个 `TemplateResponse` 对象。

`request`

An `HttpRequest` instance.

`template`

一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

```
Deprecated since version 1.8:

template 以前只接受一个Template。
```

`context`

一个 `dict` ，包含要添加到模板上下文中的值。 它默认是一个空的字典。

```
Deprecated since version 1.8:

context 以前只接受一个Context。
```

`content_type`

HTTP `Content-Type` 头部包含的值，包含MIME 类型和字符集的编码。如果指定 `content_type` ，则使用它的值。否则，使用 `DEFAULT_CONTENT_TYPE` 。

`status`

响应的HTTP 状态码。

`current_app`

包含当前视图的应用。更多信息，参见带命名空间的URL 解析策略。

> Deprecated since version 1.8:
>
> 废弃current_app 参数。你应该去设置request.current_app。

`charset`

响应编码使用的字符集。如果没有给出则从content_type中提取，如果提取不成功则使用 DEFAULT_CHARSET 设置。

`using`

加载模板使用的模板引擎的名称。

> Changed in Django 1.8:
>
> 添加charset 和using 参数。

渲染的过程

在 `TemplateResponse` 实例返回给客户端之前，它必须被渲染。渲染的过程采用模板和上下文变量的中间表示形式，并最终将它转换为可以发送给客户端的字节流。

三种情况下会渲染 `TemplateResponse` ：

- 当使用SimpleTemplateResponse.render() 方法显示渲染TemplateResponse 实例的时候。
- 当通过给response.content 赋值显式设置响应内容的时候。
- 在应用模板响应中间件之后，应用响应中间件之前。

`TemplateResponse` 只能渲染一次。 `SimpleTemplateResponse.render()` 的第一次调用设置响应的内容；以后的响应不会改变响应的内容。

然而，当显式给 `response.content` 赋值时，修改会始终生效。如果你想强制重新渲染内容，你可以重新计算渲染的内容并手工赋值给响应的内容：

```
# Set up a rendered TemplateResponse
>>> from django.template.response import TemplateResponse
>>> t = TemplateResponse(request, 'original.html', {})
>>> t.render()
>>> print(t.content)
Original content

# Re-rendering doesn't change content
>>> t.template_name = 'new.html'
>>> t.render()
>>> print(t.content)
Original content

# Assigning content does change, no render() call required
>>> t.content = t.rendered_content
>>> print(t.content)
New content
```

渲染后的回调函数

某些操作 —— 例如缓存 —— 不可以在没有渲染的模板上执行。它们必须在完整的渲染后的模板上执行。

如果你正在使用中间件，解决办法很容易。中间件提供多种在从视图退出时处理响应的机会。如果你向响应中间件添加一些行为，它们将保证在模板渲染之后执行。

然而，如果正在使用装饰器，就不会有这样的机会。装饰器中定义的行为会立即执行。

为了补偿这一点（以及其它类似的使用情形） `TemplateResponse` 允许你注册在渲染完成时调用的回调函数。使用这个回调函数，你可以延迟某些关键的处理直到你可以保证渲染后的内容是可以访问的。

要定义渲染后的回调函数，只需定义一个接收一个响应作为参数的函数并将这个函数注册到模板响应中：

```
from django.template.response import TemplateResponse

def my_render_callback(response):
    # Do content-sensitive processing
    do_post_processing()

def my_view(request):
    # Create a response
    response = TemplateResponse(request, 'mytemplate.html', {})
    # Register the callback
    response.add_post_render_callback(my_render_callback)
    # Return the response
    return response
```

`my_render_callback()` 将在 `mytemplate.html` 渲染之后调用，并将被传递一个 `TemplateResponse` 实例作为参数。

如果模板已经渲染，回调函数将立即执行。

使用TemplateResponse 和 SimpleTemplateResponse

`TemplateResponse` 对象和普通的 `django.http.HttpResponse` 一样可以用于任何地方。它可以用来作为 `render()` 和 `render_to_response()` 的另外一种选择。

例如，下面这个简单的视图使用一个简单模板和包含查询集的上下文返回一个 `TemplateResponse` ：

```
from django.template.response import TemplateResponse

def blog_index(request):
    return TemplateResponse(request, 'entry_list.html', {'entrie
s': Entry.objects.all()})
```

> 译者：Django 文档协作翻译小组，原文：TemplateResponse objects。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

TemplateResponse 和 SimpleTemplateResponse

标准的 `HttpResponse` 对象是静态的结构。在构造的时候提供给它们一个渲染好的内容，但是当内容改变时它们却不能很容易地完成相应的改变。

然而，有时候允许装饰器或者中间件在响应构造之后修改它是很有用的。例如，你可能想改变使用的模板，或者添加额外的数据到上下文中。

TemplateResponse 提供了实现这一点的方法。与基本的 `HttpResponse` 对象不同，TemplateResponse 对象会记住视图提供的模板和上下文的详细信息来计算响应。响应的最终结果在后来的响应处理过程中直到需要时才计算。

SimpleTemplateResponse 对象

class `SimpleTemplateResponse` [source]

属性

`SimpleTemplateResponse.template_name`

渲染的模板的名称。接收一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

例如： `['foo.html', 'path/to/bar.html']`

Deprecated since version 1.8: `template_name` 以前只接受一个 `Template` 。

`SimpleTemplateResponse.context_data`

渲染模板时用到的上下文数据。它必须是一个 `dict` 。

例如： `{'foo': 123}`

Deprecated since version 1.8: `context_data` 以前只接受一个 `Context` 。

`SimpleTemplateResponse.rendered_content` [source]

使用当前的模板和上下文数据渲染出来的响应内容。

`SimpleTemplateResponse.is_rendered` [source]

一个布尔值，表示响应内容是否已经渲染。

方法

`SimpleTemplateResponse.__init__` (*template*, *context=None*, *content_type=None*, *status=None*, *charset=None*, *using=None*)[source]

使用给定的模板、上下文、Content-Type、HTTP 状态和字符集实例化一个 `TemplateResponse` 对象。

`request`

An `HttpRequest` instance.

`template`

一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

Deprecated since version 1.8: `template` 以前只接受一个 `Template` 。

`context`

一个 `dict` ，包含要添加到模板上下文中的值。 它默认是一个空的字典。

Deprecated since version 1.8: `context` 以前只接受一个 `Context` 。

`content_type`

HTTP `Content-Type` 头部包含的值，包含MIME 类型和字符集的编码。如果指定 `content_type` ，则使用它的值。否则，使用 `DEFAULT_CONTENT_TYPE` 。

`status`

响应的HTTP 状态码。

`current_app`

包含当前视图的应用。更多信息，参见带命名空间的*URL* 解析策略。

Deprecated since version 1.8: 废弃 `current_app` 参数。你应该去设置 `request.current_app` 。

`charset`

响应编码使用的字符集。如果没有给出则从 `content_type` 中提取，如果提取不成功则使用 `DEFAULT_CHARSET` 设置。

`using`

加载模板使用的模板引擎的 `名称` 。

Changed in Django 1.8:

添加 `charset` 和 `using` 参数。

渲染的过程

在 `TemplateResponse` 实例返回给客户端之前，它必须被渲染。渲染的过程采用模板和上下文变量的中间表示形式，并最终将它转换为可以发送给客户端的字节流。

有三种情况会渲染 `TemplateResponse` ：

- 使用 `SimpleTemplateResponse.render()` 方法显式渲染 `TemplateResponse` 实例的时候。
- 通过给 `response.content` 赋值显式设置响应内容的时候。
- 传递给模板响应中间件之后，响应中间件之前。

`TemplateResponse` 只能渲染一次。`SimpleTemplateResponse.render()` 的第一次调用设置响应的内容；以后的响应不会改变响应的内容。

然而，当显式给 `response.content` 赋值时，修改会始终生效。如果你想强制重新渲染内容，你可以重新计算渲染的内容并手工赋值给响应的内容：

```
# Set up a rendered TemplateResponse
>>> from django.template.response import TemplateResponse
>>> t = TemplateResponse(request, 'original.html', {})
>>> t.render()
>>> print(t.content)
Original content

# Re-rendering doesn't change content
>>> t.template_name = 'new.html'
>>> t.render()
>>> print(t.content)
Original content

# Assigning content does change, no render() call required
>>> t.content = t.rendered_content
>>> print(t.content)
New content
```

渲染后的回调函数

某些操作 —— 例如缓存 —— 不可以在没有渲染的模板上执行。它们必须在完整的渲染后的模板上执行。

如果你正在使用中间件，解决办法很容易。中间件提供多种在从视图退出时处理响应的机会。如果你向响应中间件添加一些行为，它们将保证在模板渲染之后执行。

然而，如果正在使用装饰器，就不会有这样的机会。装饰器中定义的行为会立即执行。

为了补偿这一点（以及其它类似的使用情形） `TemplateResponse` 允许你注册在渲染完成时调用的回调函数。使用这个回调函数，你可以延迟某些关键的处理直到你可以保证渲染后的内容是可以访问的。

要定义渲染后的回调函数，只需定义一个接收一个响应作为参数的函数并将这个函数注册到模板响应中：

```
from django.template.response import TemplateResponse

def my_render_callback(response):
    # Do content-sensitive processing
    do_post_processing()

def my_view(request):
    # Create a response
    response = TemplateResponse(request, 'mytemplate.html', {})
    # Register the callback
    response.add_post_render_callback(my_render_callback)
    # Return the response
    return response
```

`my_render_callback()` 将在 `mytemplate.html` 渲染之后调用，并将被传递一个 `TemplateResponse` 实例作为参数。

如果模板已经渲染，回调函数将立即执行。

使用TemplateResponse 和 SimpleTemplateResponse

`TemplateResponse` 对象和普通的 `django.http.HttpResponse` 一样可以用于任何地方。它可以用来作为 `render()` 和 `render_to_response()` 的另外一种选择。

例如，下面这个简单的视图使用一个简单模板和包含查询集的上下文返回一个 `TemplateResponse` ：

```
from django.template.response import TemplateResponse

def blog_index(request):
    return TemplateResponse(request, 'entry_list.html', {'entrie
s': Entry.objects.all()})
```

文件上传

文件上传

当Django在处理文件上传的时候，文件数据被保存在 `request. FILES` （更多关于 `request` 对象的信息 请查看 请求和响应对象）。这篇文档阐述了文件如何上传到内存和硬盘，以及如何自定义默认的行为。

> 警告
>
> 允许任意用户上传文件是存在安全隐患的。更多细节请在用户上传的内容中查看有关安全指导的话题。

基本的文件上传

考虑一个简单的表单，它含有一个 `FileField` ：

```
# In forms.py...
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

处理这个表单的视图会在 `request` 中接受到上传文件的数据。 `FILES` 是个字典，它包含每个 `FileField` 的键 （或者 `ImageField` ， `FileField` 的子类）。这样的话就可以用 `request.FILES['file']` 来存放表单中的这些数据了。

注意 `request.FILES` 会仅仅包含数据，如果请求方法为 `POST` ，并且发送请求的 `<form>` 拥有 `enctype="multipart/form-data"` 属性。否则 `request.FILES` 为空。

大多数情况下，你会简单地从 `request` 向表单中传递数据，就像绑定上传文件到表单描述的那样。这样类似于：

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render_to_response('upload.html', {'form': form})
```

注意我们必须向表单的构造器中传递 `request.FILES` 。这是文件数据绑定到表单的方法。

这里是一个普遍的方法，可能你会采用它来处理上传文件：

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

遍历 `UploadedFile.chunks()` ，而不是使用 `read()` ，能确保大文件并不会占用系统过多的内存。

`UploadedFile` 对象也拥有一些其他的方法和属性；完整参考请见UploadedFile。

使用模型处理上传文件

如果你在 `Model` 上使用 `FileField` 保存文件，使用 `ModelForm` 可以让这个操作更加容易。调用 `form.save()` 的时候，文件对象会保存在相应的 `FileField` 的 `upload_to` 参数指定的地方。

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = ModelFormWithFileField(request.POST, request.FILE
S)
        if form.is_valid():
            # file is saved
            form.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = ModelFormWithFileField()
    return render(request, 'upload.html', {'form': form})
```

如果你手动构造一个对象，你可以简单地把文件对象从 `request.FILE` 赋值给模型：

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm
from .models import ModelWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FIL
ES['file'])
            instance.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})
```

上传处理器

当用户上传一个文件的时候，Django会把文件数据传递给上传处理器 – 一个小型的类，会在文件数据上传时处理它。上传处理器在 `FILE_UPLOAD_HANDLERS` 中定义，默认为：

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler",)
```

`MemoryFileUploadHandler` 和 `TemporaryFileUploadHandler` 一起提供了Django的默认文件上传行为，将小文件读取到内存中，大文件放置在磁盘中。

你可以编写自定义的处理器，来定制Django如何处理文件。例如，你可以使用自定义处理器来限制用户级别的配额，在运行中压缩数据，渲染进度条，甚至是向另一个储存位置直接发送数据，而不把它存到本地。关于如何自定义或者完全替换处理器的行为，详见编写自定义的上传处理器。

上传数据在哪里储存

在你保存上传文件之前，数据需要储存在某个地方。

通常，如果上传文件小于2.5MB，Django会把整个内容存到内存。这意味着，文件的保存仅仅涉及到从内存读取和写到磁盘，所以非常快。

但是，如果上传的文件很大，Django会把它写入一个临时文件，储存在你系统的临时目录中。在类Unix的平台下，你可以认为Django生成了一个文件，名称类似于 `/tmp/tmpzfp6I6.upload` 。如果上传的文件足够大，你可以观察到文件大小的增长，由于Django向磁盘写入数据。

这些特定值 – 2.5 MB， `/tmp` ，以及其它 -- 都仅仅是"合理的默认值"，它们可以自定义，这会在下一节中描述。

更改上传处理器的行为

Django的文件上传处理器的行为由一些设置控制。详见文件上传设置。

在运行中更改上传处理器

有时候一些特定的视图需要不同的上传处理器。在这种情况下，你可以通过修改 `request.upload_handlers` ，为每个请求覆盖上传处理器。通常，这个列表会包含 `FILE_UPLOAD_HANDLERS` 提供的上传处理器，但是你可以把它修改为其它列表。

例如，假设你编写了 `ProgressBarUploadHandler` ，它会在上传过程中向某类AJAX控件提供反馈。你可以像这样将它添加到你的上传处理器中：

```
request.upload_handlers.insert(0, ProgressBarUploadHandler())
```

在这中情况下你可能想要使用 `list.insert()` （而不是 `append()` ），因为进度条处理器需要在任何其他处理器 之前执行。要记住，多个上传处理器是按顺序执行的。

如果你想要完全替换上传处理器，你可以赋值一个新的列表：

```
request.upload_handlers = [ProgressBarUploadHandler()]
```

> 注意
>
> 你只可以在访问 `request.POST` 或者 `request.FILES` 之前修改上传处理器-- 在上传处理工作执行之后再修改上传处理就毫无意义了。如果你在读取 `request.FILES` 之后尝试修改 `request.upload_handlers` ，Django会抛出异常。
>
> 所以，你应该在你的视图中尽早修改上传处理器。
>
> `CsrfViewMiddleware` 也会访问 `request.POST` ，它是默认开启的。意思是你需要在你的视图中使用 `csrf_exempt()` ，来允许你修改上传处理器。接下来在真正处理请求的函数中，需要使用 `csrf_protect()` 。注意这意味着处理器可能会在CSRF验证完成之前开始接收上传文件。例如：

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler())
    return _upload_file_view(request)

@csrf_protect
def _upload_file_view(request):
    ... # Process request
```

File对象

`django.core.files` 模块及其子模块包含了一些用于基本文件处理的内建类。

File类

`class File(file_object)`

`File` 类是Python file 对象的一个简单封装，并带有Django特定的附加功能。需要表示文件的时候，Django内部会使用这个类。

File对象拥有下列属性和方法：

`name`

含有 `MEDIA_ROOT` 相对路径的文件名称。

`size`

文件的字节数。

`file`

这个类所封装的，原生的file 对象。

`mode`

文件的读写模式。

`open([mode=None])`

打开或者重新打开文件（同时会执行 `File.seek(0)` ）。 `mode` 参数的值和Python内建的 `open()` 相同。

重新打开一个文件时，无论文件原先以什么模式打开， `mode` 都会覆盖； `None` 的意思是以原先的模式重新打开。

`read([num_bytes=None])`

读取文件内容。可选的 `size` 参数是要读的字节数；没有指定的话，文件会一直读到结尾。

`__iter__()`

迭代整个文件，并且每次生成一行。

```
Changed in Django 1.8:

File现在使用[通用的换行符](https://www.python.org/dev/peps/pep-0278
)。以下字符会识别为换行符：Unix换行符'\n'，WIndows换行符'\r\n'，以及Maci
ntosh旧式换行符'\r'。
```

`chunks([chunk_size=None])`

迭代整个文件，并生成指定大小的一部分内容。 `chunk_size` 默认为64 KB。

处理大文件时这会非常有用，因为这样可以把他们从磁盘中读取出来，而避免将整个文件存到内存中。

`multiple_chunks([chunk_size=None])`

如果文件足够大，需要按照提供的 `chunk_size` 切分成几个部分来访问到所有内容，则返回 `True` 。

`write([content])`

将指定的内容字符串写到文件。取决于底层的储存系统，写入的内容在调用 `close()` 之前可能不会完全提交。

`close()`

关闭文件。

除了这些列出的方法， `File` 暴露了 `file` 对象的以下属性和方法： `encoding` , `fileno` , `flush` , `isatty` , `newlines` , `read` , `readinto` , `readlines` , `seek` , `softspace` , `tell` , `truncate` , `writelines` , `xreadlines` 。

ContentFile类

`class ContentFile(File)[source]`

`ContentFile` 类继承自 `File` ，但是并不像 `File` 那样，它操作字符串的内容（也支持字节集），而不是一个实际的文件。例如：

```
from __future__ import unicode_literals
from django.core.files.base import ContentFile

f1 = ContentFile("esta sentencia está en español")
f2 = ContentFile(b"these are bytes")
```

ImageFile类

`class ImageFile(file_object)[source]`

Django特地为图像提供了这个内建类。 `django.core.files.images.ImageFile` 继承了 `File` 的所有属性和方法，并且额外提供了以下的属性：

`width`

图像的像素单位宽度。

`height`

图像的像素单位高度。

附加到对象的文件的额外方法

任何关联到一个对象（比如下面的 `Car.photo` ）的 `File` 都会有一些额外的方法：

`File.save(name, content[, save=True])`

以提供的文件名和内容保存一个新文件。这样不会替换已存在的文件，但是会创建新的文件，并且更新对象来指向它。如果 `save` 为 `True` ，模型的 `save()` 方法会在文件保存之后调用。这就是说，下面两行：

```
>>> car.photo.save('myphoto.jpg', content, save=False)
>>> car.save()
```

等价于：

```
>>> car.photo.save('myphoto.jpg', content, save=True)
```

要注意 `content` 参数必须是 `File` 或者 `File` 的子类的实例，比如 `ContentFile` 。

`File.delete([save=True])`

从模型实例中移除文件，并且删除内部的文件。如果 `save` 是 `True` ，模型的 `save()` 方法会在文件删除之后调用。

> 译者：Django 文档协作翻译小组，原文：File objects。
>
>
>
>

文件储存API

获取当前的储存类

Django提供了两个便捷的方法来获取当前的储存类：

`class DefaultStorage[source]`

`DefaultStorage` 提供对当前的默认储存系统的延迟访问，像 `DEFAULT_FILE_STORAGE` 中定义的那样。 `DefaultStorage` 内部使用了 `get_storage_class()` 。

`get_storage_class([import_path=None])[source]`

返回实现储存API的类或者模块。

当没有带着 `import_path` 参数调用的时候， `get_storage_class` 会返回当前默认的储存系统，像 `DEFAULT_FILE_STORAGE` 中定义的那样。如果提供了 `import_path` ， `get_storage_class` 会尝试从提供的路径导入类或者模块，并且如果成功的话返回它。如果导入不成功会抛出异常。

FileSystemStorage类

`class FileSystemStorage([location=None, base_url=None, file_permiss`

`FileSystemStorage` 类在本地文件系统上实现了基本的文件存储功能。它继承自 `Storage` ，并且提供父类的所有公共方法的实现。

`location`

储存文件的目录的绝对路径。默认为 `MEDIA_ROOT` 设置的值。

`base_url`

在当前位置提供文件储存的URL。默认为 `MEDIA_URL` 设置的值。

`file_permissions_mode`

文件系统的许可，当文件保存时会接收到它。默认为 `FILE_UPLOAD_PERMISSIONS` 。

```
New in Django 1.7:

新增了file_permissions_mode属性。之前，文件总是会接收到FILE_UPLOAD_PERMISSIONS许可。
```

`directory_permissions_mode`

文件系统的许可，当目录保存时会接收到它。默认为 `FILE_UPLOAD_DIRECTORY_PERMISSIONS` 。

> New in Django 1.7:
>
> 新增了directory_permissions_mode属性。之前，目录总是会接收到FILE_UPLOAD_DIRECTORY_PERMISSIONS许可。

> 注意
>
> `FileSystemStorage.delete()` 在提供的文件名称不存在的时候并不会抛出任何异常。

Storage类

`class Storage[source]`

`Storage` 类为文件的存储提供了标准化的API，并带有一系列默认行为，所有其它的文件存储系统可以按需继承或者复写它们。

> 注意
>
> 对于返回原生 `datetime` 对象的方法，所使用的有效时区为 `os.environ['TZ']` 的当前值。要注意它总是可以通过Django的 `TIME_ZONE` 来设置。

`accessed_time(name)[source]`

返回包含文件的最后访问时间的原生 `datetime` 对象。对于不能够返回最后访问时间的储存系统，会抛出 `NotImplementedError` 异常。

`created_time(name)[source]`

返回包含文件创建时间的原生 `datetime` 对象。对于不能够返回创建时间的储存系统，会抛出 `NotImplementedError` 异常。

`delete(name)[source]`

删除 `name` 引用的文件。如果目标储存系统不支持删除操作，会抛出 `NotImplementedError` 异常。

`exists(name)[source]`

如果提供的名称所引用的文件在文件系统中存在，则返回 `True` ，否则如果这个名称可用于新文件，返回 `False` 。

`get_available_name(name, max_length=None)[source]`

返回基于 `name` 参数的文件名称，它在目标储存系统中可用于写入新的内容。

如果提供了 `max_length` ，文件名称长度不会超过它。如果不能找到可用的、唯一的文件名称，会抛出 `SuspiciousFileOperation` 异常。

如果 `name` 命名的文件已存在，一个下划线加上随机7个数字或字母的字符串会添加到文件名称的末尾，扩展名之前。

```
Changed in Django 1.7:

之前，下划线和一位数字（比如"_1"，"_2"，以及其他）会添加到文件名称的末尾，
直到目标目录中发现了可用的名称。一些恶意的用户会利用这一确定性的算法来进行do
s攻击。这一变化也在1.6.6， 1.5.9， 和 1.4.14中出现。
```

```
Changed in Django 1.8:

新增了max_length参数。
```

`get_valid_name(name)[source]`

返回基于 `name` 参数的文件名称，它适用于目标储存系统。

`listdir(path)[source]`

列出特定目录的所有内容，返回一个包含2元组的列表；第一个元素是目录，第二个是文件。对于不能够提供列表功能的储存系统，抛出 `NotImplementedError` 异常。

`modified_time(name)[source]`

返回包含最后修改时间的原生 `datetime` 对象。对于不能够返回最后修改时间的储存系统，抛出 `NotImplementedError` 异常。

`open(name, mode='rb')[source]`

通过提供的 `name` 打开文件。注意虽然返回的文件确保为 `File` 对象，但可能实际上是它的子类。在远程文件储存的情况下，这意味着读写操作会非常慢，所以警告一下。

`path(name)[source]`

本地文件系统的路径，文件可以用Python标准的 `open()` 在里面打开。对于不能从本地文件系统访问的储存系统，抛出 `NotImplementedError` 异常。

`save(name, content, max_length=None)[source]`

使用储存系统来保存一个新文件，最好带有特定的名称。如果名称为 `name` 的文件已存在，储存系统会按需修改文件名称来获取一个唯一的名称。返回被储存文件的实际名称。

max_length参数会传递给 `get_available_name()` 。

`content` 参数必须为 `django.core.files.File` 或者 `File` 子类的实例。

> Changed in Django 1.8:
>
> 新增了max_length参数。

`size(name)[source]`

返回 `name` 所引用的文件的总大小，以字节为单位。对于不能够返回文件大小的储存系统，抛出 `NotImplementedError` 异常。

`url(name)[source]`

返回URL，通过它可以访问到 `name` 所引用的文件。对于不支持通过URL访问的储存系统，抛出 `NotImplementedError` 异常。

> 译者：Django 文档协作翻译小组，原文：Storage API。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

管理文件

这篇文档描述了Django为那些用户上传文件准备的文件访问API。底层的API足够通用，你可以使用为其它目的来使用它们。如果你想要处理静态文件（JS，CSS，以及其他），参见管理静态文件（CSS和图像）。

通常，Django使用 `MEDIA_ROOT` 和 `MEDIA_URL` 设置在本地储存文件。下面的例子假设你使用这些默认值。

然而，Django提供了一些方法来编写自定义的 文件储存系统，允许你完全自定义Django在哪里以及如何储存文件。这篇文档的另一部分描述了这些储存系统如何工作。

在模型中使用文件

当你使用 `FileField` 或者 `ImageField` 的时候，Django为你提供了一系列的API用来处理文件。

考虑下面的模型，它使用 `ImageField` 来储存一张照片：

```
from django.db import models

class Car(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to='cars')
```

任何 `Car` 的实例都有一个 `photo` 字段，你可以通过它来获取附加图片的详细信息：

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: chevy.jpg>
>>> car.photo.name
'cars/chevy.jpg'
>>> car.photo.path
'/media/cars/chevy.jpg'
>>> car.photo.url
'http://media.example.com/cars/chevy.jpg'
```

例子中的 `car.photo` 对象是一个 `File` 对象，这意味着它拥有下面描述的所有方法和属性。

> 注意
>
> 文件保存是数据库模型保存的一部分，所以磁盘上真实的文件名在模型保存之前并不可靠。

例如，你可以通过设置文件的 `name` 属性为一个和文件储存位置（ `MEDIA_ROOT` ，如果你使用默认的 `FileSystemStorage` ）相关的路径，来修改文件名称。

```
>>> import os
>>> from django.conf import settings
>>> initial_path = car.photo.path
>>> car.photo.name = 'cars/chevy_ii.jpg'
>>> new_path = settings.MEDIA_ROOT + car.photo.name
>>> # Move the file on the filesystem
>>> os.rename(initial_path, new_path)
>>> car.save()
>>> car.photo.path
'/media/cars/chevy_ii.jpg'
>>> car.photo.path == new_path
True
```

File

当Django需要表示一个文件的时候，它在内部使用 `django.core.files.File` 实例。这个对象是 Python 内建文件对象的一个简单封装，并带有一些Django特定的附加功能。

大多数情况你可以简单地使用Django提供给你的 `File` 对象（例如像上面那样把文件附加到模型，或者是上传的文件）。

如果你需要自行构造一个 `File` 对象，最简单的方法是使用Python内建的 `file` 对象来创建一个：

```
>>> from django.core.files import File

# Create a Python file object using open()
>>> f = open('/tmp/hello.world', 'w')
>>> myfile = File(f)
```

现在你可以使用 `File` 类的任何文档中记录的属性和方法了。

注意这种方法创建的文件并不会自动关闭。以下步骤可以用于自动关闭文件：

```
>>> from django.core.files import File

# Create a Python file object using open() and the with statemen
t
>>> with open('/tmp/hello.world', 'w') as f:
...     myfile = File(f)
...     myfile.write('Hello World')
...
>>> myfile.closed
True
>>> f.closed
True
```

在处理大量对象的循环中访问文件字段时，关闭文件极其重要。如果文件在访问之后没有手动关闭，会有消耗完文件描述符的风险。这可能导致如下错误：

```
IOError: [Errno 24] Too many open files
```

文件储存

在背后，Django需要决定在哪里以及如何将文件储存到文件系统。这是一个对象，它实际上理解一些东西，比如文件系统，打开和读取文件，以及其他。

Django的默认文件储存由 `DEFAULT_FILE_STORAGE` 设置提供。如果你没有显式提供一个储存系统，就会使用它。

关于内建的默认文件储存系统的细节，请参见下面一节。另外，关于编写你自己的文件储存系统的一些信息，请见编写自定义的文件系统。

储存对象

大多数情况你可能并不想使用 `File` 对象（它向文件提供适当的存储功能），你可以直接使用文件储存系统。你可以创建一些自定义文件储存类的实例，或者 – 大多数情况更加有用的 – 你可以使用全局的默认储存系统：

```
>>> from django.core.files.storage import default_storage
>>> from django.core.files.base import ContentFile

>>> path = default_storage.save('/path/to/file', ContentFile('ne
w content'))
>>> path
'/path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

关于文件储存API，参见 文件储存API。

内建的文件系统储存类

Django自带了 `django.core.files.storage.FileSystemStorage` 类，它实现了基本的本地文件系统中的文件储存。

例如，下面的代码会在 `/media/photos` 目录下储存上传的文件，无论 `MEDIA_ROOT` 设置是什么：

```
from django.db import models
from django.core.files.storage import FileSystemStorage

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

自定义储存系统 以相同方式工作：你可以把它们作为 `storage` 参数传递给 `FileField` 。

> 译者：Django 文档协作翻译小组，原文：Managing files。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写自定义存储系统

如果你需要提供自定义文件存储 – 一个普遍的例子是在某个远程系统上储存文件 – 你可以通过定义一个自定义的储存类来实现。你需要遵循以下步骤：

1. 你的自定义储存类必须是 `django.core.files.storage.Storage` 的子类：

```
from django.core.files.storage import Storage

class MyStorage(Storage):
    ...
```

2. Django必须能够不带任何参数来实例化你的储存类。这意味着任何设置都应该从 `django.conf.settings` 中获取。

```
from django.conf import settings
from django.core.files.storage import Storage

class MyStorage(Storage):
    def __init__(self, option=None):
        if not option:
            option = settings.CUSTOM_STORAGE_OPTIONS
        ...
```

3. 你的储存类必须实现 `_open()` 和 `_save()` 方法，以及任何适合于你的储存类的其它方法。更多这类方法请见下文。

另外，如果你的类提供本地文件存储，它必须覆写 `path()` 方法。

4. 你的储存类必须是 可以析构的，所以它在迁移中的一个字段上使用的时候可以被序列化。只要你的字段拥有自己可以序列化的参数，你就可以为它使用 `django.utils.deconstruct.deconstructible` 类装饰器（这也是Django用在 `FileSystemStorage` 上的东西）。

默认情况下，下面的方法会抛出 `NotImplementedError` 异常，并且必须覆写它们。

- Storage.delete()
- Storage.exists()
- Storage.listdir()
- Storage.size()
- Storage.url()

然而要注意，并不是这些方法全部都需要，可以故意省略一些。可以不必实现每个方法而仍然能拥有一个可以工作的储存类。

比如，如果在特定的储存后端中，列出内容的开销比较大，你可以决定不实现 `Storage.listdir` 。

另一个例子是只处理写入文件的后端。这种情况下，你不需要实现上面的任意一种方法。

根本上来说，需要实现哪种方法取决于你。如果不去实现一些方法，你会得到一个不完整（可能是不能用的）的接口。

你也会经常想要使用特意为自定义储存对象设计的钩子。它们是：

`_open(name, mode='rb')`

必需的。

被 `Storage.open()` 调用，这是储存类用于打开文件的实际工具。它必须返回 `File` 对象，在大多数情况下，你会想要返回一些子类，它们实现了后端储存系统特定的逻辑。

`_save(name, content)`

被 `Storage.save()` 调用。 `name` 必须事先通过 `get_valid_name()` 和 `get_available_name()` 过滤，并且 `content` 自己必须是一个 `File` 对象。

应该返回被保存文件的真实名称（通常是传进来的 `name` ，但是如果储存需要修改文件名称，则返回新的名称来代替）。

`get_valid_name(name)`

返回适用于当前储存系统的文件名。传递给该方法的 `name` 参数是发送给服务器的原始文件名称，并移除了所有目录信息。你可以覆写这个方法，来自定义非标准的字符将会如何转换为安全的文件名称。

`Storage` 提供的代码只会保留原始文件名中的数字和字母字符、英文句号和下划线，并移除其它字符。

`get_available_name(name, max_length=None)`

返回在储存系统中可用的文件名称，可能会顾及到提供的文件名称。传给这个方法的name参数需要事先过滤为储存系统有效的文件名称，根据上面描述的 `get_valid_name()` 方法。

如果提供了 `max_length` ，文件名称长度不会超过它。如果不能找到可用的、唯一的文件名称，会抛出 `SuspiciousFileOperation` 异常。

如果 `name` 命名的文件已存在，一个下划线加上随机7个数字或字母的字符串会添加到文件名称的末尾，扩展名之前。

```
Changed in Django 1.7:

之前，下划线和一位数字（比如"_1"，"_2"，以及其他）会添加到文件名称的末尾，直到目标目录中发现了可用的名称。一些恶意的用户会利用这一确定性的算法来进行dos攻击。 这一变化也在1.6.6， 1.5.9， 和 1.4.14中出现。
```

```
Changed in Django 1.8:

新增了max_length参数。
```

自定义储存系统 以相同方式工作：你可以把它们作为 `storage` 参数传递给 `FileField` 。

> 译者：Django 文档协作翻译小组，原文：Custom storage。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

基于类的视图

基于类的视图

视图是一个可调用对象，它接收一个请求然后返回一个响应。这个可调用对象可以不只是函数，Django 提供一些可以用作视图的类。它们允许你结构化你的视图并且利用继承和混合重用代码。后面我们将介绍一些用于简单任务的通用视图，但你可能想要设计自己的可重用视图的结构以适合你的使用场景。完整的细节，请参见基于类的视图的参考文档。

- 基于类的视图简介
- 内建的基于类的通用视图
- 使用基于类的视图处理表单
- 使用混合来扩展视图类

基本的示例

Django 提供基本的视图类，它们适用于广泛的应用。所有的视图类继承自 `View` 类，它负责连接视图到URL、HTTP 方法调度和其它简单的功能。 `RedirectView` 用于简单的HTTP 重定向， `TemplateView` 扩展基类来渲染模板。

在URLconf 中的简单用法

使用通用视图最简单的方法是在URLconf 中创建它们。如果你只是修改基于类的视图的一些简单属性，你可以将它们直接传递给 `as_view()` 方法调用：

```
from django.conf.urls import url
from django.views.generic import TemplateView

urlpatterns = [
    url(r'^about/', TemplateView.as_view(template_name="about.ht
ml")),
]
```

传递给 `as_view()` 的参数将覆盖类中的属性。在这个例子中，我们设置 `TemplateView` 的 `template_name` 。可以使用类似的方法覆盖 `RedirectView` 的 `url` 属性。

子类化通用视图

第二种，功能更强一点的使用通用视图的方式是继承一个已经存在的视图并在子类中覆盖其属性（例如 `template_name` ）或方法（例如 `get_context_data` ）以提供新的值或方法。例如，考虑只显示一个模板 `about.html` 的视图。Django 有一个通用视图 `TemplateView` 来做这件事，所以我们可以简单地子类化它，并覆盖模板的名称：

```
# some_app/views.py
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

然后我们只需要添加这个新的视图到我们的URLconf 中。 `TemplateView` 是一个类不是一个函数，所以我们将URL 指向类的 `as_view()` 方法，它让基于类的视图提供一个类似函数的入口：

```
# urls.py
from django.conf.urls import url
from some_app.views import AboutView

urlpatterns = [
    url(r'^about/', AboutView.as_view()),
]
```

关于如何使用内建的通用视图的更多信息，参考下一主题通用的基于类的视图。

支持其它HTTP 方法

假设有人想通过HTTP 访问我们的书库，它使用视图作为API。这个API 客户端将随时连接并下载自上次访问以来新出版的书籍的数据。如果没有新的书籍，仍然从数据库中获取书籍、渲染一个完整的响应并发送给客户端将是对CPU 和带宽的浪费。如果有个API 用于查询书籍最新发布的时间将会更好。

我们在URLconf 中映射URL 到书籍列表视图：

```
from django.conf.urls import url
from books.views import BookListView

urlpatterns = [
    url(r'^books/$', BookListView.as_view()),
]
```

下面是这个视图：

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book

class BookListView(ListView):
    model = Book

    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest('publication_date
')
        response = HttpResponse('')
        # RFC 1123 date format
        response['Last-Modified'] = last_book.publication_date.s
trftime('%a, %d %b %Y %H:%M:%S GMT')
        return response
```

如果该视图从GET 请求访问，将在响应中返回一个普通而简单的对象列表（使用 `book_list.html` 模板）。但如果客户端发出一个 `HEAD` 请求，响应将具有一个空的响应体而 `Last-Modified` 头部会指示最新发布的书籍的时间。基于这个信息，客户端可以下载或不下载完整的对象列表。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

基于类的内建通用视图

编写Web应用可能是单调的，因为你需要不断的重复某一种模式。 Django尝试从model和 template层移除一些单调的情况，但是Web开发者依然会在view（视图）层经历这种厌烦。

Django的通用视图被开发用来消除这一痛苦。它们采用某些常见的习语和在开发过程中发现的模式然后把它们抽象出来，以便你能够写更少的代码快速的实现基础的视图。

我们能够识别一些基础的任务，比如展示对象的列表，以及编写代码来展示任何对象的 列表。此外，有问题的模型可以作为一个额外的参数传递到URLconf中。

Django通过通用视图来完成下面一些功能：

- 为单一的对象展示列表和一个详细页面。 如果我们创建一个应用来管理会议，那么 一个 TalkListView (讨论列表视图)和一个 RegisteredUserListView （注册用户列表视图）就是列表视图的一个例子。一个单独的讨论信息页面就是我们称 之为 "详细" 视图的例子。
- 在年/月/日归档页面，以及详细页面和“最后发表”页面中，展示以数据库为基础的对象。 允许用户创建，更新和删除对象 -- 以授权或者无需授权的方式。

总的来说，这些视图提供了一些简单的接口来完成开发者遇到的大多数的常见任务。

扩展通用视图

使用通用视图可以极大的提高开发速度，是毫无疑问的。 然而在大多数工程中，总会遇到通用视图无法满足需求的时候。的确，大多数来自Django开发新手 的问题是如何能使得通用视图的使用范围更广。

这是通用视图在1.3发布中被重新设计的原因之一 - 之前，它们仅仅是一些函数视图加上 一列令人疑惑的选项；现在，比起传递大量的配置到URLconf中，更推荐的扩展通用视图的 方法是子类化它们，并且重写它们的属性或者方法。

这就是说，通用视图有一些限制。如果你将你的视图实现为通用视图的子类，你就会发现这样能够更有效地编写你想要的代码，使用你自己的基于类或功能的视图。

在一些三方的应用中，有更多通用视图的示例，或者你可以自己按需编写。

对象的通用视图

TemplateView确实很有用，但是当你需要 呈现你数据库中的内容时Django的通用视图才眞的会脱颖而出。因为这是如此常见 的任务，Django提供了一大把内置的通用视图，使生成对象的展示列表和详细视图 的变得极其容易。

让我们来看一下这些通用视图中的"对象列表"视图。

我们将使用下面的模型：

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

现在我们需要定义一个视图：

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

最后将视图解析到你的url上：

```
# urls.py
from django.conf.urls import url
from books.views import PublisherList

urlpatterns = [
    url(r'^publishers/$', PublisherList.as_view()),
]
```

上面就是所有我们需要写的Python代码了。

> 注意
>
> 所以，当（例如）DjangoTemplates后端的APP_DIRS选项在TEMPLATES中设置为True时，模板的位置应该为：/path/to/project/books/templates/books/publisher_list.html。

这个模板将会依据于一个上下文(context)来渲染，这个context包含一个名为 object_list 包含所有publisher对象的变量。一个非常简单的模板可能看起来像下面这样：

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

这确实就是全部代码了。 所有通用视图中有趣的特性来自于修改被传递到通用视图中的"信息" 字典。generic views reference文档详细 介绍了通用视图以及它的选项；本篇文档剩余的部分将会介绍自定义以及扩展通用 视图的常见方法。

编写“友好的”模板上下文

你可能已经注意到了，我们在publisher列表的例子中把所有的publisher对象 放到 object_list 变量中。虽然这能正常工作，但这对模板作者并不是 "友好的"。他们只需要知道在这里要处理publishers就行了。

因此，如果你在处理一个模型(model)对象，这对你来说已经足够了。 当你处理 一个object或者queryset时，Django能够使用你定义对象显示用的自述名（verbose name，或者复数的自述名，对于对象列表）来填充上下文(context)。提供添加到默认的 object_list 实体中，但是包含完全相同的数据，例如publisher_list。

如果自述名(或者复数的自述名) 仍然不能很好的符合要求，你 可以手动的设置上下文(context)变量的名字。在一个通用视图上的context_object_name属性指定了要使用的定了上下文变量：

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
    context_object_name = 'my_favorite_publishers'
```

提供一个有用的context_object_name总是个好主意。和你一起工作的设计 模板的同事会感谢你的。

添加额外的上下文

多数时候，你只是需要展示一些额外的信息而不是提供一些通用视图。 比如，考虑到每个publisher 详细页面上的图书列表的展示。DetailView通用视图提供了一个publisher对象给context，但是我们如何在模板中添加附加信息呢？

答案是派生DetailView，并且在get_context_data方法中提供你自己的实现。默认的实现只是简单的 给模板添加了要展示的对象，但是你这可以这样覆写来展示更多信息：

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(
**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

> 注意
>
> 通常来说，get_context_data会将当前类中的上下文数据，合并到所有超类中的上下文数据。要在你自己想要改变上下文的类中保持这一行为，你应该确保在超类中调用了get_context_data。如果没有任意两个类尝试定义相同的键，会返回异常的结果。然而，如果任何一个类尝试在超类持有一个键的情况下覆写它（在调用超类之后），这个类的任何子类都需要显式于超类之后设置它，如果你想要确保他们覆写了所有超类的话。如果你有这个麻烦，复查你视图中的方法调用顺序。

查看对象的子集

现在让我们来近距离查看下我们一直在用的 model参数。model参数指定了视图在哪个数据库模型之上进行操作，这适用于所有的需要 操作一个单独的对象或者一个对象集合的通用视图。然而，model参数并不是唯一能够指明视图要基于哪个对象进行操作的方法 -- 你同样可以使用queryset参数来指定一个对象列表：

```
from django.views.generic import DetailView
from books.models import Publisher

class PublisherDetail(DetailView):

    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

指定model = Publisher等价于快速声明的queryset = Publisher.objects.all()。然而，通过使用queryset来定义一个过滤的对象列表，你可以更加详细 的了解哪些对象将会被显示的视图中(参见执行查询来获取更多关于查询集对象的更对信息，以及参见 基于类的视图参考来获取全部 细节)。

我们可能想要对图书列表按照出版日期进行排序来选择一个简单的例子，并且把 最近的放到前面：

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

这是个非常简单的列子，但是它很好的诠释了处理思路。 当然，你通常想做的不仅仅只是 对对象列表进行排序。如果你想要展现某个出版商的所有图书列表，你可以使用 同样的手法：

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):

    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='Acme Publish
ing')
    template_name = 'books/acme_list.html'
```

注意，除了经过过滤之后的查询集，一起定义的还有我们自定义的模板名称。如果我们不这么做，通过视图会使用和 "vanilla" 对象列表名称一样的模板，这可 能不是我们想要的。

另外需要注意，这并不是处理特定出版商的图书的非常优雅的方法。 如果我们 要创建另外一个出版商页面，我们需要添加另外几行代码到URLconf中，并且再多几个 出版商就会觉得这么做不合理。我们会在下一个章节处理这个问题。

> 注意
>
> 如果你在访问 /books/acme/时出现404错误，检查确保你确实有一个名字为“ACME Publishing”的出版商。通用视图在这种情况下拥有一个allow_empty的参数。详见基于类的视图参考。

动态过滤

另一个普遍的需求是在给定的列表页面中根据URL中的关键字来过滤对象。 前面我们把出版 商的名字硬编码到URLconf中，但是如果我们想要编写一个视图来展示任何publisher的所有 图书，应该如何处理？

相当方便的是， ListView 有一个get_queryset() 方法来供我们重写。在之前，它只是返回一个queryset属性值，但是现在我们可以添加更多的逻辑。

让这种方式能够工作的关键点，在于当类视图被调用时，各种有用的对象被存储在self上；同request()(self.request)一样，其中包含了从URLconf中获取到的位置参数(self.args)和基于名字的参数(self.kwargs)(关键字参数)。

这里，我们拥有一个带有一组供捕获的参数的URLconf：

```
# urls.py
from django.conf.urls import url
from books.views import PublisherBookList

urlpatterns = [
    url(r'^books/([\w-]+)/$', PublisherBookList.as_view()),
]
```

接着，我们编写了PublisherBookList视图::

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.
args[0])
        return Book.objects.filter(publisher=self.publisher)
```

如你所见，在queryset区域添加更多的逻辑非常容易；如果我们想的话，我们可以使用self.request.user来过滤当前用户，或者添加其他更复杂的逻辑。

同时我们可以把出版商添加到上下文中，这样我们就可以在模板中使用它：

```
# ...

def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super(PublisherBookList, self).get_context_data(**
kwargs)
    # Add in the publisher
    context['publisher'] = self.publisher
    return context
```

执行额外的工作

我们需要考虑的最后的共同模式在调用通用视图之前或者之后会引起额外的开销。

想象一下，在我们的Author对象上有一个last_accessed字段，这个字段用来 跟踪某人最后一次查看了这个作者的时间。

```
# models.py
from django.db import models

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')
    last_accessed = models.DateTimeField()
```

通用的DetailView类，当然不知道关于这个字段的事情，但我们可以很容易 再次编写一个自定义的视图，来保持这个字段的更新。

首先，我们需要添加作者详情页的代码配置到URLconf中，指向自定义的视图：

```
from django.conf.urls import url
from books.views import AuthorDetailView

urlpatterns = [
    #...
    url(r'^authors/(?P<pk>[0-9]+)/$', AuthorDetailView.as_view()
, name='author-detail'),
]
```

然后，编写我们新的视图 -- get_object是用来获取对象的方法 -- 因此我们简单的 重写它并封装调用：

```
from django.views.generic import DetailView
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):

    queryset = Author.objects.all()

    def get_object(self):
        # Call the superclass
        object = super(AuthorDetailView, self).get_object()
        # Record the last accessed date
        object.last_accessed = timezone.now()
        object.save()
        # Return the object
        return object
```

> 注意
>
> 这里URLconf使用参数组的名字pk - 这个名字是DetailView用来查找主键的值的默认名称，其中主键用于过滤查询集。
>
> 如果你想要调用参数组的其它方法，你可以在视图上设置pk_url_kwarg。详见DetailView参考。

使用基于类的视图处理表单

表单的处理通常有3 个步骤：

- 初始的的GET （空白或预填充的表单）
- 带有非法数据的POST（通常重新显示表单和错误信息）
- 带有合法数据的POST（处理数据并重定向）

你自己实现这些功能经常导致许多重复的样本代码（参见在视图中使用表单）。为了避免这点，Django 提供一系列的通用的基于类的视图用于表单的处理。

基本的表单

根据一个简单的联系人表单：

```
#forms.py

from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

可以使用 `FormView` 来构造其视图：

```
#views.py

from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been PO
STed.
        # It should return an HttpResponse.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

注：

- `FormView` 继承 `TemplateResponseMixin` 所以这里可以使用 `template_name` 。
- `form_valid()` 的默认实现只是简单地重定向到 `success_url` 。

模型的表单

通用视图在于模型一起工作时会真正光芒四射。这些通用的视图将自动创建一个 ModelForm，只要它们能知道使用哪一个模型类：

- 如果给出 `model` 属性，则使用该模型类。
- 如果 `get_object()` 返回一个对象，则使用该对象的类。
- 如果给出 `queryset` ，则使用该查询集的模型。

模型表单提供一个 `form_valid()` 的实现，它自动保存模型。如果你有特殊的需求，可以覆盖它；参见下面的例子。

你甚至不需要为 `CreateView` 和 `UpdateView` 提供 `success_url` —— 如果存在它们将使用模型对象的 `get_absolute_url()` 。

如果你想使用一个自定义的 `ModelForm` （例如添加额外的验证），只需简单地在你的视图上设置 `form_class` 。

> 注
>
> 当指定一个自定义的表单类时，你必须指定模型，即使 `form_class` 可能是一个 `ModelForm` 。

首先我们需要添加 `get_absolute_url()` 到我们的 `Author` 类中：

```
#models.py

from django.core.urlresolvers import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

然后我们可以使用 `CreateView` 机器伙伴来做实际的工作。注意这里我们是如何配置通用的基于类的视图的；我们自己没有写任何逻辑：

```
#views.py

from django.views.generic.edit import CreateView, UpdateView, De
leteView
from django.core.urlresolvers import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

> 注
>
> 这里我们必须使用 `reverse_lazy()` 而不是 `reverse`，因为在该文件导入时URL 还没有加载。

`fields` 属性的工作方式与 `ModelForm` 的内部 `Meta` 类的 `fields` 属性相同。除非你用另外一种方式定义表单类，该属性是必须的，如果没有将引发一个 `ImproperlyConfigured` 异常。

如果你同时指定 `fields` 和 `form_class` 属性，将引发一个 `ImproperlyConfigured` 异常。

```
Changed in Django 1.8:

省略fields 属性在以前是允许的，但是导致表单带有模型的所有字段。
```

```
Changed in Django 1.8:

以前，如果fields 和form_class 两个都指定，会默默地忽略 fields。
```

最后，我我们来将这些新的视图放到URLconf 中：

```
#urls.py

from django.conf.urls import url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = [
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author_ad
d'),
    url(r'author/(?P<pk>[0-9]+)/$', AuthorUpdate.as_view(), name
='author_update'),
    url(r'author/(?P<pk>[0-9]+)/delete/$', AuthorDelete.as_view(
), name='author_delete'),
]
```

> 注
>
> 这些表单继承`SingleObjectTemplateResponseMixin`，它使用`template_name_suffix`并基于模型来构造`template_name`。
>
> 在这个例子中：
>
> - `CreateView` 和 `UpdateView` 使用 `myapp/author_form.html`
> - `DeleteView` 使用 `myapp/author_confirm_delete.html`
>
> 如果你希望分开`CreateView` 和 `UpdateView` 的模板，你可以设置你的视图类的`template_name` 或 `template_name_suffix`。

模型和request.user

为了跟踪使用CreateView 创建一个对象的用户，你可以使用一个自定义的ModelForm 来实现这点。首先，向模型添加外键关联：

```
#models.py

from django.contrib.auth.models import User
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User)

    # ...
```

在这个视图中，请确保你没有将`created_by` 包含进要编辑的字段列表，并覆盖`form_valid()` 来添加这个用户：

```
#views.py

from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

注意，你需要使用 `login_required()` 来装饰这个视图，或者在 `form_valid()` 中处理未认证的用户。

AJAX 示例

下面是一个简单的实例，展示你可以如何实现一个表单，使它可以同时为AJAX 请求和‘普通的’表单POST 工作：

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    """
    Mixin to add AJAX support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """
    def form_invalid(self, form):
        response = super(AjaxableResponseMixin, self).form_inval
id(form)
        if self.request.is_ajax():
            return JsonResponse(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method
because
        # it might do some processing (in the case of CreateView
, it will
        # call form.save() for example).
        response = super(AjaxableResponseMixin, self).form_valid
(form)
        if self.request.is_ajax():
            data = {
                'pk': self.object.pk,
            }
            return JsonResponse(data)
        else:
            return response

class AuthorCreate(AjaxableResponseMixin, CreateView):
    model = Author
    fields = ['name']
```

> 译者：Django 文档协作翻译小组，原文：Built-in editing views。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Mixin

注意

这是一个进阶的话题。需要建立在了解 基于类的视图的基础上。

Django的基于类的视图提供了许多功能，但是你可能只想使用其中的一部分。例如，你想编写一个视图，它渲染模板来响应HTTP，但是你用不了 `TemplateView` ；或者你只需要对 `POST` 请求渲染一个模板，而 `GET` 请求做一些其它的事情。 虽然你可以直接使用 `TemplateResponse` ，但是这将导致重复的代码。

由于这些原因，Django 提供许多Mixin，它们提供更细致的功能。例如，渲染模板封装在 `TemplateResponseMixin` 中。Django 参考手册包含所有*Mixin* 的完整文档。

Context 和TemplateResponse

在基于类的视图中使用模板具有一致的接口，有两个Mixin 起了核心的作用。

`TemplateResponseMixin`

返回 `TemplateResponse` 的每个视图都将调用 `render_to_response()` 方法，这个方法由 `TemplateResponseMixin` 提供。大部分时候，这个方法会隐式调用（例如，它会被 `TemplateView` 和 `DetailView` 中实现的 `get()` 方法调用）；如果你不想通过Django 的模板渲染响应，那么你可以覆盖它，虽然通常不需要这样。其示例用法请参见*JSONResponseMixin* 示例。

`render_to_response()` 本身调用 `get_template_names()` ，它默认查找类视图的 `template_name` ； 其它两个Mixin（ `SingleObjectTemplateResponseMixin` 和 `MultipleObjectTemplateResponseMixin` ）覆盖了这个方法，以在处理实际的对象时能提供更灵活的默认行为。

`ContextMixin`

需要Context 数据的每个内建视图，例如渲染模板的视图（包括 `TemplateResponseMixin` ），都应该以关键字参数调用 `get_context_data()` ，以确保它们想要的数据在里面。 `get_context_data()` 返回一个字典；在 `ContextMixin` 中，它只是简单地返回它的关键字参数，通常会覆盖这个方法来向字典中添加更多的成员。

构建Django 的基于类的通用视图函数

让我们看下Django 的两个通用的基于类的视图是如何通过互不相关的Mixin 构建的。我们将考虑 `DetailView` ，它渲染一个对象的“详细”视图，和 `ListView` ，它渲染一个对象列表，通常来自一个查询集，需要时还会分页。这将会向我们接收四个Mixin，这些Mixin 在用到单个或多个Django对象时非常有用。

在通用的编辑视图（ `FormView` 和模型相关的视图 `CreateView` 、 `UpdateView` 和 `DeleteView` ）和基于日期的通用视图中都会涉及到Minxin。它们在*Mixin 参考文档*中讲述。

DetailView：用于单个Django 对象

为了显示一个对象的详细信息，我们通常需要做两件事情：查询对象然后利用合适的模板和包含该对象的Context 生成 `TemplateResponse` 。

为了获得对象， `DetailView` 依赖 `SingleObjectMixin` ，它提供一个 `get_object()` 方法，这个方法基于请求的URL 获取对象（它查找URLconf 中声明的 `pk` 和 `slug` 关键字参数，然后从视图的 `model` 属性或 `queryset` 属性查询对象）。 `SingleObjectMixin` 还覆盖 `get_context_data()` ，这个方法在Django 所有的内建的基于类的视图中都有用到，用来给模板的渲染提供Context 数据。

然后，为了生成 `TemplateResponse` ， `DetailView` 使用 `SingleObjectTemplateResponseMixin` ，它扩展自 `TemplateResponseMixin` 并覆盖上文讨论过的 `get_template_names()` 。实际上，它提供比较复杂的选项集合，但是大部分人用到的主要的一个是 `<app_label>/<model_name>_detail.html` 。 `_detail` 部分可以通过设置子类的 `template_name_suffix` 来改变。（例如，通用的编辑视图 使用 `_form` 来创建和更新视图，用 `_confirm_delete` 来删除视图）。

ListView：用于多个Django 对象

显示对象的列表和上面的步骤大体相同：我们需要一个对象的列表（可能是分页形式的），这通常是一个 `QuerySet` ，然后我们需要利用合适的模板和对象列表生成一个 `TemplateResponse` 。

为了获取对象， `ListView` 使用 `MultipleObjectMixin` ，它提供 `get_queryset()` 和 `paginate_queryset()` 两种方法。与 `SingleObjectMixin` 不同，不需要根据URL 中关键字参数来获得查询集，默认将使用视图类的 `queryset` 或 `model` 属性。通常需要覆盖 `get_queryset()` 以动态获取不同的对象，例如根据当前的用户或排除打算在将来提交的博客。

`MultipleObjectMixin` 还覆盖 `get_context_data()` 以包含合适的Context 变量用于分页（如果禁止分页，则提供一些假的）。这个方法依赖传递给它的关键字参数 `object_list` ， `ListView` 会负责准备好这个参数。

为了生成 `TemplateResponse` ， `ListView` 然后使用 `MultipleObjectTemplateResponseMixin` ；与上面的 `SingleObjectTemplateResponseMixin` 类似，它覆盖 `get_template_names()` 来提供 一系列的选项 ，而最常用到的是 `<app_label>/<model_name>_list.html` ，其中 `_list` 部分同样由 `template_name_suffix` 属性设置。（基于日期的通用视图使用 `_archive` 、 `_archive_year` 等等这样的后缀来针对各种基于日期的列表视图使用不同的模板）。

使用Django 的基于类的视图的Mixin

既然我们已经看到Django 通用的基于类的视图时如何使用Mixin，让我们在看看其它组合它们的方式。当然，我们仍将它们与内建的基于类的视图或其它通用的基于类的视图组合，但是对于Django 提供的便利性你将解决一些更加罕见的问题。

警告

不是所有的Mixin 都可以一起使用，也不是所有的基于类的视图都可以与其它Mixin 一起使用。这里我们展示的是可以工作的几个例子；如果你需要其它功能，那么你必须考虑不同类之间的属性和方法的相互作用，以及方法解析顺序将如何影响方法调用的版本和顺序。

Django 的基于类的视图 和基于类的视图的*Mixin* 的文档将帮助你理解在不同的类和Mixin 之间那些属性和方法可能引起冲突。

如果有担心，通常最好退避并基于 `View` 或 `TemplateView` ，或者可能的话加上 `SingleObjectMixin` 和 `MultipleObjectMixin` 。虽然你可能最终会编写更多的代码，但是对于后来的人更容易理解，而且你自己也少了份担心。（当然，你始终可以深入探究Django 中基于类的通用视图的具体实现以获取如何处理出现的问题的灵感）。

SingleObjectMixin 与View 一起使用

如果你想编写一个简单的基于类的视图，它只响应 `POST` ，我们将子类化 `View` 并在子类中只编写一个 `post()` 方法。但是，如果我们想处理一个由URL 标识的特定对象，我们将需要 `SingleObjectMixin` 提供的功能。

我们将使用在通用的基于类的视图简介 中用到的 `Author` 模型做演示。

views.py

```
from django.http import HttpResponseForbidden, HttpResponseRedir
ect
from django.core.urlresolvers import reverse
from django.views.generic import View
from django.views.generic.detail import SingleObjectMixin
from books.models import Author

class RecordInterest(SingleObjectMixin, View):
    """Records the current user's interest in an author."""
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(reverse('author-detail', kwa
rgs={'pk': self.object.pk}))
```

实际应用中，你的对象可能以键-值的方式保存而不是保存在关系数据库中，所以我们不考虑这点。使用 `SingleObjectMixin` 的视图唯一需要担心的是在哪里查询我们感兴趣的Author，而它会用一个简单的 `self.get_object()` 调用实现。其它的所有事情都有该Mixin 帮我们处理。

我们可以将它这样放入URL 中，非常简单：

urls.py

```
from django.conf.urls import url
from books.views import RecordInterest

urlpatterns = [
    #...
    url(r'^author/(?P<pk>[0-9]+)/interest//span>, RecordInterest
.as_view(), name='author-interest'),
]
```

注意 `pk` 命名组，`get_object()` 将用它来查询 `Author` 实例。你还可以使用slug，或者 `SingleObjectMixin` 的其它功能。

SingleObjectMixin 与ListView 一起使用

`ListView` 提供内建的分页，但是可能你分页的列表中每个对象都与另外一个对象（通过一个外键）关联。在我们的Publishing 例子中，你可能想通过一个特定的Publisher 分页所有的Book。

一种方法是组合 `ListView` 和 `SingleObjectMixin` ，这样分页的Book 列表的查询集能够与找到的单个Publisher 对象关联。为了实现这点，我们需要两个不同的查询集：

`Book` queryset for use by `ListView`

Since we have access to the `Publisher` whose books we want to list, we simply override `get_queryset()` and use the `Publisher` 's *reverse foreign key manager*.

`Publisher` queryset for use in `get_object()`

We'll rely on the default implementation of `get_object()` to fetch the correct `Publisher` object. However, we need to explicitly pass a `queryset` argument because otherwise the default implementation of `get_object()` would call `get_queryset()` which we have overridden to return `Book` objects instead of `Publisher` ones.

注

我们必须仔细考虑 `get_context_data()` 。因为 `SingleObjectMixin` 和 `ListView` 都会将Context 数据的 `context_object_name` 下，我们必须显式确保 `Publisher` 位于Context 数据中。 `ListView` 将为我们添加合适的 `page_obj` 和 `paginator` ，只要我们记住调用 `super()` 。

现在，我们可以编写一个新的 `PublisherDetail` ：

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher

class PublisherDetail(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Publisher.objects
.all())
        return super(PublisherDetail, self).get(request, *args,
**kwargs)

    def get_context_data(self, **kwargs):
        context = super(PublisherDetail, self).get_context_data(
**kwargs)
        context['publisher'] = self.object
        return context

    def get_queryset(self):
        return self.object.book_set.all()
```

注意我们 在 `get()` 方法里设置了 `self.object` ，这样我们就可以在后面的 `get_context_data()` 和 `get_queryset()` 方法里再次用到它. 如果不设置 `template_name` ,那模板会指向默认的 `ListView` 所选择的模板, 也就是 `"books/book_list.html"` ，因为这个模板是书目的一个列表; 但 `ListView` 对于该类继承了 `SingleObjectMixin` 这个类是一无所知的,所以不会对使用 `Publisher` 来查看视图有任何反应.

`paginate_by` 是每页显示几条数据的意思，这里设的比较小，是因为这样你就不用造一堆数据才能看到分页的效果了！下面是你想要的模板:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publisher {{ publisher.name }}</h2>

    <ol>
      {% for book in page_obj %}
        <li>{{ book.title }}</li>
      {% endfor %}
    </ol>

     class="pagination">
         class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number
}}">previous</a>
            {% endif %}

             class="current">
                Page {{ page_obj.number }} of {{ paginator.num_p
ages }}.


            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">
next</a>
            {% endif %}


{% endblock %}
```

避免让事情复杂化

通常情况下你只在需要相关功能时才会使用 `TemplateResponseMixin` 和 `SingleObjectMixin` 这两个类。如上所示，只要加点儿小心，你甚至可以把 `SingleObjectMixin` 和 `ListView` 结合在一起来使用. 但是这么搞可能会让事情变得有点复杂，作为一个好的原则：

提示：

你的视图扩展应该仅仅使用那些来自于同一组通用基类的view或者mixins。如: *detail, list*, *editing* 和 date. 例如：把 `TemplateView` (内建视图)和 `MultipleObjectMixin` (通用列表)整合在一起是极好的, 但是若想把 `SingleObjectMixin` (generic detail) 和 `MultipleObjectMixin` (generic list)整合在一起就有麻烦啦！

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. 首先，我们来看一下如何把 `DetailView` 和 `FormMixin` 结合起来，实现 `POST` 一个 Django 表单 到相同URL，这样我们就可以用 `DetailView` 来显示具体对象了.

使用 FormMixin 与 DetailView

想想我们之前合用 `View` 和 `SingleObjectMixin` 的例子. 我们想要记录用户对哪些作者感兴趣; 也就是说我们想让用户发表说为什么喜欢这些作者的信息。同样的，我们假设这些数据并没有存放在关系数据库里，而是存在另外一个奥妙之地（其实这里不用关心具体存放到了哪里）。

要实现这一点，自然而然就要设计一个? `Form` ，让用户把相关信息通过浏览器发送到Django后台。 另外，我们要巧用REST方法,这样我们就可以用相同的URL来显示作者和捕捉来自用户的消息了。 让我们重写 `AuthorDetailView` 来实现它。

We'll keep the `GET` handling from `DetailView`, although we'll have to add a `Form` into the context data so we can render it in the template. We'll also want to pull in form processing from `FormMixin`, and write a bit of code so that on `POST` the form gets called appropriately.

Note

We use `FormMixin` and implement `post()` ourselves rather than try to mix `DetailView` with `FormView` (which provides a suitable `post()` already) because both of the views implement `get()`, and things would get much more confusing.

Our new `AuthorDetail` looks like this:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseForbidden
from django.core.urlresolvers import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDetail(FormMixin, DetailView):
    model = Author
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.objec
t.pk})

    def get_context_data(self, **kwargs):
        context = super(AuthorDetail, self).get_context_data(**k
wargs)
        context['form'] = self.get_form()
        return context

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()
        self.object = self.get_object()
        form = self.get_form()
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        # Here, we would record the user's interest using the me
ssage
        # passed in form.cleaned_data['message']
        return super(AuthorDetail, self).form_valid(form)
```

`get_success_url()` is just providing somewhere to redirect to, which gets used in the default implementation of `form_valid()`. We have to provide our own `post()` as noted earlier, and override `get_context_data()` to make the `Form` available in the context data.

优化方案

It should be obvious that the number of subtle interactions between `FormMixin` and `DetailView` is already testing our ability to manage things. 你不太可能会去想自己写这种类的。

In this case, it would be fairly easy to just write the `post()` method yourself, keeping `DetailView` as the only generic functionality, although writing `Form` handling code involves a lot of duplication.

Alternatively, it would still be easier than the above approach to have a separate view for processing the form, which could use `FormView` distinct from `DetailView` without concerns.

其他可选的方案

What we're really trying to do here is to use two different class based views from the same URL. So why not do just that? We have a very clear division here: `GET` requests should get the `DetailView` (with the `Form` added to the context data), and `POST` requests should get the `FormView` . Let's set up those views first.

The `AuthorDisplay` view is almost the same as *when we first introduced AuthorDetail*; we have to write our own `get_context_data()` to make the `AuthorInterestForm` available to the template. We'll skip the `get_object()` override from before for clarity:

```
from django.views.generic import DetailView
from django import forms
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDisplay(DetailView):
    model = Author

    def get_context_data(self, **kwargs):
        context = super(AuthorDisplay, self).get_context_data(**
kwargs)
        context['form'] = AuthorInterestForm()
        return context
```

? `AuthorInterest` 是一个简单的 `FormView` ，但是我们不得不把 `SingleObjectMixin` 引入进来，这样我们才能定位我们评论的作者，并且我们还要记得设置 `template_name` 来确保form出错时使用? `GET` 会渲染到? `AuthorDisplay` 相同的模板：

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseForbidden
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin

class AuthorInterest(SingleObjectMixin, FormView):
    template_name = 'books/author_detail.html'
    form_class = AuthorInterestForm
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()
        self.object = self.get_object()
        return super(AuthorInterest, self).post(request, *args,
**kwargs)

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.objec
t.pk})
```

Finally we bring this together in a new `AuthorDetail` view. We already know that calling `as_view()` on a class-based view gives us something that behaves exactly like a function based view, so we can do that at the point we choose between the two subviews.

You can of course pass through keyword arguments to `as_view()` in the same way you would in your URLconf, such as if you wanted the `AuthorInterest` behavior to also appear at another URL but using a different template:

```
from django.views.generic import View

class AuthorDetail(View):

    def get(self, request, *args, **kwargs):
        view = AuthorDisplay.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterest.as_view()
        return view(request, *args, **kwargs)
```

This approach can also be used with any other generic class-based views or your own class-based views inheriting directly from `View` or `TemplateView`, as it keeps the different views as separate as possible.

返回HTML 以外的内容

基于类的视图在同一件事需要实现多次的时候非常有优势。假设你正在编写API，每个视图应该返回JSON 而不是渲染后的HTML。

我们可以创建一个Mixin 类来处理JSON 的转换，并将它用于所有的视图。

例如，一个简单的JSON Mixin 可能像这样：

```
from django.http import JsonResponse

class JSONResponseMixin(object):
    """
 A mixin that can be used to render a JSON response.
 """
    def render_to_json_response(self, context, **response_kwargs
):
        """
 Returns a JSON response, transforming 'context' to make the pay
load.
 """
        return JsonResponse(
            self.get_data(context),
            **response_kwargs
        )

    def get_data(self, context):
        """
 Returns an object that will be serialized as JSON by json.dumps
().
 """
        # Note: This is *EXTREMELY* naive; in reality, you'll ne
ed
        # to do much more complex handling to ensure that arbitr
ary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return context
```

注

查看序列化*Django* 对象 的文档，其中有如何正确转换Django 模型和查询集到JSON 的更多信息。

该Mixin 提供一个 `render_to_json_response()` 方法，它与 `render_to_response()` 的参数相同。要使用它，我们只需要将它与 `TemplateView` 组合，并覆盖 `render_to_response()` 来调用 `render_to_json_response()` ：

```
from django.views.generic import TemplateView

class JSONView(JSONResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_
kwargs)
```

同样地，我们可以将我们的Mixin 与某个通用的视图一起使用。我们可以实现自己的 `DetailView` 版本，将 `JSONResponseMixin` 和 `django.views.generic.detail.BaseDetailView` 组合– (the `DetailView` before template rendering behavior has been mixed in):

```
from django.views.generic.detail import BaseDetailView

class JSONDetailView(JSONResponseMixin, BaseDetailView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_
kwargs)
```

这个视图可以和其它 `DetailView` 一样使用，它们的行为完全相同 —— 除了响应的格式之外。

如果你想更进一步，你可以组合 `DetailView` 的子类，它根据HTTP 请求的某个属性既能够返回HTML 又能够返回JSON 内容，例如查询参数或HTTP 头部。这只需将 `JSONResponseMixin` 和 `SingleObjectTemplateResponseMixin` 组合，并覆盖 `render_to_response()` 的实现以根据用户请求的响应类型进行正确的渲染：

```
from django.views.generic.detail import SingleObjectTemplateResp
onseMixin

class HybridDetailView(JSONResponseMixin, SingleObjectTemplateRe
sponseMixin, BaseDetailView):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get('format') == 'json':
            return self.render_to_json_response(context)
        else:
            return super(HybridDetailView, self).render_to_respo
nse(context)
```

由于Python 解析方法重载的方式，`super(HybridDetailView, self).render_to_response(context)` 调用将以调用 `TemplateResponseMixin` 的 `render_to_response()` 实现结束。

内建基于类的视图的API

基于类的视图的API 参考。另请参见基于类的视图 的简介。

说明

由基于类的视图处理的每个请求都具有一个独立的状态；所以，在实例中保存状态变量是安全的（例如， `self.foo = 3` 是线程安全的操作）。

基于类的视图在URL 模式中的部署使用 `as_view()` 类方法：

```
urlpatterns = [
    url(r'^view/$', MyView.as_view(size=42)),
]
```

> 视图参数的线程安全性
>
> 传递给视图的参数在视图的每个实例之间共享。这表示不应该使用列表、字典或其它可变对象作为视图的参数。如果你真这么做而且对共享的对象做过修改，某个用户的行为可能对后面访问同一个视图的用户产生影响。

传递给 `as_view()` 的参数将赋值给服务请求的实例。利用前面的例子，这表示对 `MyView` 的每个请求都可以使用 `self.size` 。参数必须对应于在类中已经存在的属性（ `hasattr` 检查可以返回 `True` ）。

基础视图 VS. 通用视图

基于类的基础视图可以认为是父视图，它们可以直接使用或者继承它们。它们不能满足项目中所有的需求，在这种情况下有Mixin 可以扩展基础视图的功能。

Django 的通用视图建立在基础视图之上，用于作为经常用到的功能的快捷方式，例如显示对象的详细信息。它们提炼视图开发中常见的风格和模式并将它们抽象，这样你可以快速编写常见的视图而不用重复你自己。

大部分通常视图需要`queryset` 键，它是一个`查询集` 实例；关于`查询集` 对象的更多信息，请参见执行查询。

> 译者：Django 文档协作翻译小组，原文：API reference。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

基于类的通用视图 —— 索引

这里的索引提供基于类的视图的另外一种组织形式。对于每个视图，在类继承树中有效的属性和方法都显示在该视图的下方。按照行为进行组织的文档，参见基于类的视图。

简单的通用视图

View

`class View`

属性（以及访问它们的方法）：

- `http_method_names`

方法

- `as_view()`
- `dispatch()`
- `head()`
- `http_method_not_allowed()`

TemplateView

`class TemplateView`

属性（以及访问它们的方法）：

- `content_type`
- `http_method_names`
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `head()`
- `http_method_not_allowed()`
- `render_to_response()`

RedirectView

`class RedirectView`

属性（以及访问它们的方法）：

- `http_method_names`
- `pattern_name`
- `permanent`
- `query_string`
- `url` [`get_redirect_url()`]

方法

- `as_view()`
- `delete()`
- `dispatch()`
- `get()`
- `head()`
- `http_method_not_allowed()`
- `options()`
- `post()`
- `put()`

明细视图

DetailView

`class DetailView`

属性（以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `http_method_names`
- `model`
- `pk_url_kwarg`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `render_to_response()`

清单视图

ListView

`class ListView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `http_method_names`
- `model`
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

编辑视图

FormView

`class FormView`

属性（以及访问它们的方法）：

- `content_type`
- `form_class` [`get_form_class()`]
- `http_method_names`
- `initial` [`get_initial()`]
- `prefix` [`get_prefix()`]
- `response_class` [`render_to_response()`]
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]

方法

- `as_view()`
- `dispatch()`
- `form_invalid()`
- `form_valid()`
- `get()`
- `get_context_data()`
- `get_form()`
- `get_form_kwargs()`
- `http_method_not_allowed()`
- `post()`
- `put()`

CreateView

`class CreateView`

属性（以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `fields`
- `form_class` [`get_form_class()`]
- `http_method_names`
- `initial` [`get_initial()`]
- `model`
- `pk_url_kwarg`
- `prefix` [`get_prefix()`]
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]

- `slug_url_kwarg`
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `form_invalid()`
- `form_valid()`
- `get()`
- `get_context_data()`
- `get_form()`
- `get_form_kwargs()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `post()`
- `put()`
- `render_to_response()`

UpdateView

`class UpdateView`

属性（以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `fields`
- `form_class` [`get_form_class()`]
- `http_method_names`
- `initial` [`get_initial()`]
- `model`
- `pk_url_kwarg`
- `prefix` [`get_prefix()`]
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`

- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `form_invalid()`
- `form_valid()`
- `get()`
- `get_context_data()`
- `get_form()`
- `get_form_kwargs()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `post()`
- `put()`
- `render_to_response()`

DeleteView

`class DeleteView`

属性（以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `http_method_names`
- `model`
- `pk_url_kwarg`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `delete()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_object()`

- `head()`
- `http_method_not_allowed()`
- `post()`
- `render_to_response()`

基于日期的视图

ArchiveIndexView

`class ArchiveIndexView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`
- `model`
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

YearArchiveView

`class YearArchiveView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`
- `make_object_list` [`get_make_object_list()`]
- `model`
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

MonthArchiveView

`class MonthArchiveView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`

- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]
- `month_format` [`get_month_format()`]
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_next_month()`
- `get_paginator()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

WeekArchiveView

```
class WeekArchiveView
```

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`

- `model`
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `week` [`get_week()`]
- `week_format` [`get_week_format()`]
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

DayArchiveView

`class DayArchiveView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `day` [`get_day()`]
- `day_format` [`get_day_format()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]
- `month_format` [`get_month_format()`]

- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_next_day()`
- `get_next_month()`
- `get_paginator()`
- `get_previous_day()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

TodayArchiveView

`class TodayArchiveView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `day` [`get_day()`]
- `day_format` [`get_day_format()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]

- `month_format` [`get_month_format()`]
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_next_day()`
- `get_next_month()`
- `get_paginator()`
- `get_previous_day()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

DateDetailView

`class DateDetailView`

属性（以及访问它们的方法）：

- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `day` [`get_day()`]
- `day_format` [`get_day_format()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]

- `month_format` [`get_month_format()`]
- `pk_url_kwarg`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_next_day()`
- `get_next_month()`
- `get_object()`
- `get_previous_day()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `render_to_response()`

> 译者：Django 文档协作翻译小组，原文：Flattened index。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

高级

使用Django输出CSV

这篇文档阐述了如何通过使用Django视图动态输出CSV (Comma Separated Values)。 你可以使用Python CSV 库或者Django的模板系统来达到目的。

使用Python CSV库

Python自带了CSV库， `csv` 。在Django中使用它的关键是， `csv` 模块的CSV创建功能作用于类似于文件的对象，并且Django的 `HttpResponse` 对象就是类似于文件的对象。

这里是个例子：

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV he
ader.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="som
efilename.csv"'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C', '"Testing"', "
Here's a quote"])

    return response
```

代码和注释是不用多说的，但是一些事情需要提醒一下：

- 响应对象获得了一个特殊的MIME类型， `text/csv` 。这会告诉浏览器，文档是个CSV文件而不是HTML文件。如果你把它去掉，浏览器可能会把输出解释为HTML，会在浏览器窗口中显示一篇丑陋的、可怕的官样文章。
- 响应对象获取了附加的 `Content-Disposition` 协议头，它含有CSV文件的名称。文件名可以是任意的；你想把它叫做什么都可以。浏览器会在”另存为“对话框中使用它，或者其它。
- 钩住CSV生成API非常简单：只需要把 `response` 作为第一个参数传递给 `csv.writer` 。 `csv.writer` 函数接受一个类似于文件的对象，而 `HttpResponse` 对象正好合适。
- 对于你CSV文件的每一行，调用 `writer.writerow` ，向它传递一个可迭代的对象比如列表或者元组。
- CSV模板会为你处理引用，所以你不用担心没有转义字符串中的引号或者逗号。只需要向 `writerow()` 传递你的原始字符串，它就会执行正确的操作。

> 在Python 2中处理Unicode
>
> Python2的 `csv` 模块不支持Unicode输入。由于Django在内部使用Unicode，这意味着从一些来源比如 `HttpRequest` 读出来的字符串可能导致潜在的问题。有一些选项用于处理它：
>
> - 手动将所有Unicode对象编码为兼容的编码。
> - 使用csv模块示例章节中提供的 `UnicodeWriter` 类。
> - 使用python-unicodecsv 模块，它作为 `csv` 模块随时可用的替代方案，能够优雅地处理Unicode。
>
> 更多信息请见 `csv` 模块的Python文档。

流式传输大尺寸CSV文件

当处理生成大尺寸响应的视图时，你可能想要使用Django的 `StreamingHttpResponse` 类。例如，通过流式传输需要长时间来生成的文件，可以避免负载均衡器在服务器生成响应的时候断掉连接。

在这个例子中，我们利用Python的生成器来有效处理大尺寸CSV文件的拼接和传输：

```
import csv

from django.utils.six.moves import range
from django.http import StreamingHttpResponse

class Echo(object):
    """An object that implements just the write method of the fi
le-like
    interface.
    """
    def write(self, value):
        """Write the value by returning it, instead of storing i
n a buffer."""
        return value

def some_streaming_csv_view(request):
    """A view that streams a large CSV file."""
    # Generate a sequence of rows. The range is based on the max
imum number of
    # rows that can be handled by a single sheet in most spreads
heet
    # applications.
    rows = (["Row {}".format(idx), str(idx)] for idx in range(65
536))
    pseudo_buffer = Echo()
    writer = csv.writer(pseudo_buffer)
    response = StreamingHttpResponse((writer.writerow(row) for r
ow in rows),
                                     content_type="text/csv")
    response['Content-Disposition'] = 'attachment; filename="som
efilename.csv"'
    return response
```

使用模板系统

或者，你可以使用Django模板系统来生成CSV。比起便捷的Python `csv` 模板来说，这样比较低级，但是为了完整性，这个解决方案还是在这里展示一下。

它的想法是，传递一个项目的列表给你的模板，并且让模板在 `for` 循环中输出逗号。

这里是一个例子，它像上面一样生成相同的CSV文件：

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV he
ader.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="som
efilename.csv"'

    # The data is hard-coded here, but you could load it from a
database or
    # some other source.
    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quo
te"),
    )

    t = loader.get_template('my_template_name.txt')
    c = Context({
        'data': csv_data,
    })
    response.write(t.render(c))
    return response
```

这个例子和上一个例子之间唯一的不同就是，这个例子使用模板来加载，而不是CSV模块。代码的结果 -- 比如 `content_type='text/csv'` -- 都是相同的。

然后，创建模板 `my_template_name.txt` ，带有以下模板代码：

```
{% for row in data %}"{{ row.0|addslashes }}", "{{ row.1|addslas
hes }}", "{{ row.2|addslashes }}", "{{ row.3|addslashes }}", "{{
 row.4|addslashes }}"
{% endfor %}
```

这个模板十分基础。它仅仅遍历了提供的数据，并且对于每一行都展示了一行CSV。它使用了 `addslashes` 模板过滤器来确保没有任何引用上的问题。

其它基于文本的格式

要注意对于 CSV来说，这里并没有什么特别之处 -- 只是特定了输出格式。你可以使用这些技巧中的任何一个，来输出任何你想要的，基于文本的格式。你也可以使用相似的技巧来生成任意的二进制数据。例子请参见在Django中输出PDF。

> 译者：Django 文档协作翻译小组，原文：Generating CSV。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

使用Django输出PDF

这篇文档阐述了如何通过使用Django视图动态输出PDF。这可以通过一个出色的、开源的Python PDF库ReportLab来实现。

动态生成PDF文件的优点是，你可以为不同目的创建自定义的PDF -- 这就是说，为不同的用户或者不同的内容。

例如，Django在kusports.com上用来为那些参加March Madness比赛的人，生成自定义的，便于打印的 NCAA 锦标赛晋级表作为PDF文件。

安装ReportLab

ReportLab库在PyPI上提供。也可以下载到用户指南 （PDF文件，不是巧合）。 你可以使用 `pip` 来安装ReportLab：

```
$ pip install reportlab
```

通过在Python交互解释器中导入它来测试你的安装：

```
>>> import reportlab
```

若没有抛出任何错误，则已安装成功。

编写你的视图

使用Django动态生成PDF的关键是，ReportLab API作用于类似于文件的对象，并且Django的 `HttpResponse` 对象就是类似于文件的对象。

这里是一个 “Hello World”的例子：

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF he
aders.
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="som
efilename.pdf"'

    # Create the PDF object, using the response object as its "f
ile."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation ha
ppens.
    # See the ReportLab documentation for the full list of funct
ionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

代码和注释是不用多说的，但是一些事情需要提醒一下：

- 响应对象获得了一个特殊的MIME类型， `application/pdf` 。这会告诉浏览器，文档是个PDF文件而不是HTML文件。 如果你把它去掉，浏览器可能会把输出解释为HTML，会在浏览器窗口中显示一篇丑陋的、可怕的官样文章。
- 响应对象获取了附加的 `Content-Disposition` 协议头，它含有PDF文件的名称。 文件名可以是任意的；你想把它叫做什么都可以。浏览器会在”另存为“对话框中使用它，或者其它。
- 在这个例子中， `Content-Disposition` 协议头以 `'attachment;'` 开头。这样就强制让浏览器弹出对话框来提示或者确认，如果机器上设置了默认值要如何处理文档。如果你去掉了 `'attachment;'` ，无论什么程序或控件被设置为用于处理PDF，浏览器都会使用它。代码就像这样：

```
response['Content-Disposition'] = 'filename="somefilename.pdf"'
```

- 钩住ReportLab API 非常简单：只需要向 `canvas.Canvas` 传递 `response` 作为第一个参数。 `Canvas` 函数接受一个类似于文件的对象，而 `HttpResponse` 对象正好合适。
- 注意所有随后的PDF生成方法都在PDF对象（这个例子是p）上调用，而不是 `response` 对象上。
- 最后，在PDF文件上调用 `showPage()` 和 `save()` 非常重要。

> 注意
>
> ReportLab并不是线程安全的。一些用户报告了一些奇怪的问题，在构建生成PDF的Django视图时出现，这些视图在同一时间被很多人访问。

复杂的PDF

如果你使用ReportLab创建复杂的PDF文档，考虑使用 `io` 库作为你PDF文件的临时保存地点。这个库提供了一个类似于文件的对象接口，非常实用。这个是上面的“Hello World”示例采用 `io` 重写后的样子：

```
from io import BytesIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF he
aders.
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="som
efilename.pdf"'

    buffer = BytesIO()

    # Create the PDF object, using the BytesIO object as its "fi
le."
    p = canvas.Canvas(buffer)

    # Draw things on the PDF. Here's where the PDF generation ha
ppens.
    # See the ReportLab documentation for the full list of funct
ionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly.
    p.showPage()
    p.save()

    # Get the value of the BytesIO buffer and write it to the re
sponse.
    pdf = buffer.getvalue()
    buffer.close()
    response.write(pdf)
    return response
```

更多资源

- PDFlib与Python捆绑的另一个PDF生成库。在Django中使用它的方法和这篇文

章所阐述的相同。
- Pisa XHTML2PDF是另一个PDF生成库。Pisa自带了如何将 Pisa 集成到Django的例子。
- HTMLdoc是一个命令行脚本，它可以把HTML转换为PDF。它并没有Python接口，但是你可以使用 `system` 或者 `popen` ，在控制台中使用它，然后再Python中取回输出。

其它格式

要注意在这些例子中并没有很多PDF特定的东西 -- 只是使用了 `reportlab` 。你可以使用相似的技巧来生成任何格式，只要你可以找到对应的Python库。关于用于生成基于文本的格式的其它例子和技巧，另见使用Django输出CSV。

> 译者：Django 文档协作翻译小组，原文：Generating PDF。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

中间件

中间件

中间件是一个介入Django的请求和响应的处理过程中的钩子框架。它是一个轻量级，底层的“插件”系统，用于在全局修改Django的输入或输出。

中间件组件责任处理某些特殊的功能。例如，Django包含一个中间件组件，AuthenticationMiddleware ，使用会话将用户和请求关联。

这篇文档讲解了中间件如何工作，如何激活中间件，以及如何编写自己的中间件。Django集成了一些内置的中间件可以直接开箱即用。它们被归档在 内置中间件参考.

激活中间件

要激活一个中间件组件，需要把它添加到你Django配置文件中的MIDDLEWARE_CLASSES 列表中。

在MIDDLEWARE_CLASSES中，每一个中间件组件用字符串的方式描述：一个完整的Python全路径加上中间件的类名称。例如，使用 django-admin startproject创建工程的时候生成的默认值：

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddlew
are',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
)
```

Django的程序中，中间件不是必需的 — 只要你喜欢，MIDDLEWARE_CLASSES可以为空 — 但是强烈推荐你至少使用CommonMiddleware。

MIDDLEWARE_CLASSES中的顺序非常重要，因为一个中间件可能依赖于另外一个。例如，AuthenticationMiddleware在会话中储存已认证的用户。所以它必须在SessionMiddleware之后运行。一些关于Django中间件类的顺序的常见提示，请见Middleware ordering。

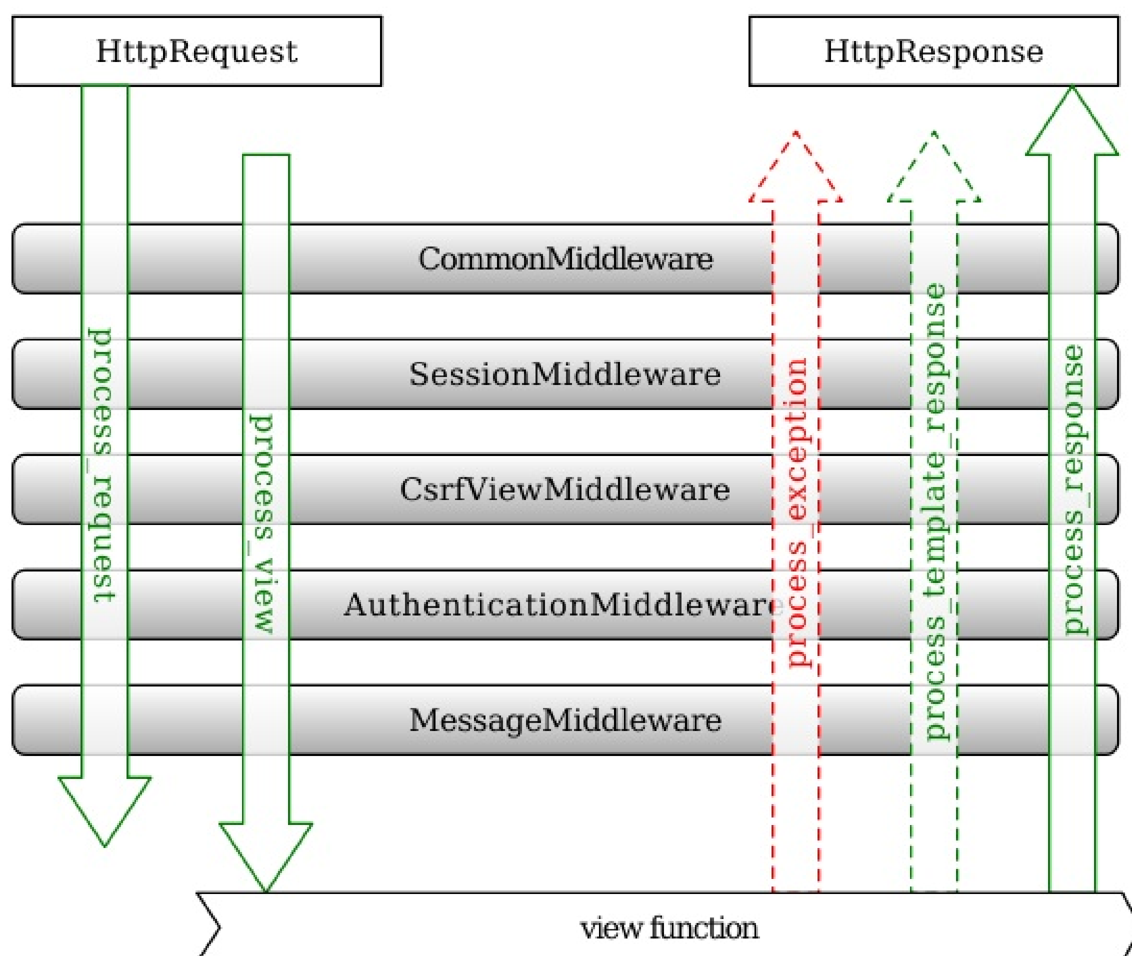
钩子和应用顺序

在请求阶段中，调用视图之前，Django会按照MIDDLEWARE_CLASSES中定义的顺序自顶向下应用中间件。会用到两个钩子：

- process_request()
- process_view()

在响应阶段中，调用视图之后，中间件会按照相反的顺序应用，自底向上。会用到三个钩子：

- process_exception()（仅当视图抛出异常的时候）
- process_template_response()（仅用于模板响应）
- process_response()



如果你愿意的话，你可以把它想象成一颗洋葱：每个中间件都是包裹视图的一层“皮”。

每个钩子的行为接下来会描述。

编写自己的中间件

编写自己的中间件很容易的。每个中间件组件是一个单独的Python的class，你可以定一个或多个下面的这些方法：

process_request

process_request(request)

request是一个HttpRequest 对象。

在Django决定执行哪个视图(view)之前，process_request()会被每次请求调用。

它应该返回一个None 或一个HttpResponse对象。如果返回 None, Django会继续处理这个请求，执行其他process_request()中间件，然后process_view()中间件显示对应的视图。如果它返回一个HttpResponse对象，Django便不再会去调用其他的请求(request), 视图(view)或其他中间件，或对应的视图；处理HttpResponse的中间件会处理任何返回的响应(response)。

process_view

process_view(request, view_func, view_args, view_kwargs)

request是一个HttpRequest对象。view_func是 Django会调用的一个Python的函数。(它确实是一个函数对象，不是函数的字符名称。) view_args是一个会被传递到视图的位置参数列表，而view_kwargs 是一个会被传递到视图的关键字参数字典。 view_args和 view_kwargs 都不包括第一个视图参数(request)。

process_view()会在Django调用视图(view)之前被调用。

它将返回None 或一个HttpResponse 对象。如果返回 None，将会继续处理这个请求，执行其他的process_view() 中间件，然后显示对应的视图。如果返回HttpResponse对象，Django就不再会去调用其他的视图（view），异常中间件（exception middleware）或对应的视图 ；它会把响应中间件应用到HttpResponse上，并返回结果。

> 注意
>
> 在中间件内部，从process_request或者process_view方法中访问request.POST或者request.REQUEST将会阻碍该中间 件之后的所有视图无法修改request的上传处理程序， 一般情况要避免这样使用。
>
> 类CsrfViewMiddleware可以被认为是个例外 ，因为它提供了csrf_exempt() 和csrf_protect()两个允许视图来精确控制 在哪个点需要开启CSRF验证。

process_template_response

process_template_response(request, response)

request是一个HttpRequest对象。response是一个TemplateResponse对象（或等价的对象），由Django视图或者中间件返回。

如果响应的实例有render()方法，process_template_response()在视图刚好执行完毕之后被调用，这表明了它是一个TemplateResponse对象（或等价的对象）。

这个方法必须返回一个实现了render方法的响应对象。它可以修改给定的response对象，通过修改 response.template_name和response.context_data或者它可以创建一个全新的 TemplateResponse或等价的对象。

你不需要显式渲染响应 —— 一旦所有的模板响应中间件被调用，响应会自动被渲染。

在一个响应的处理期间，中间件以相反的顺序运行，这包括 process_template_response()。

process_response

process_response(request, response)

request是一个HttpRequest对象。response是Django视图或者中间件返回的HttpResponse或者StreamingHttpResponse对象。

process_response()在所有响应返回浏览器之前被调用。

这个方法必须返回HttpResponse或者StreamingHttpResponse对象。它可以改变已有的response，或者创建并返回新的HttpResponse或StreamingHttpResponse对象。

不像 process_request()和process_view()方法，即使同一个中间件类中的process_request()和process_view()方法会因为前面的一个中间件返回HttpResponse而被跳过，process_response()方法总是会被调用。特别是，这意味着你的process_response()方法不能依赖于process_request()方法中的设置。

最后，记住在响应阶段中，中间件以相反的顺序被应用，自底向上。意思是定义在MIDDLEWARE_CLASSES最底下的类会最先被运行。

处理流式响应

不像HttpResponse，StreamingHttpResponse并没有content属性。所以，中间件再也不能假设所有响应都带有content属性。如果它们需要访问内容，他们必须测试是否为流式响应，并相应地调整自己的行为。

```
if response.streaming:
    response.streaming_content = wrap_streaming_content(response
.streaming_content)
else:
    response.content = alter_content(response.content)
```

> 注意
>
> 我们需要假设streaming_content可能会大到在内存中无法容纳。响应中间件可能会把它封装在新的生成器中，但是一定不要销毁它。封装一般会实现成这样：
>
> ```
> def wrap_streaming_content(content):
>     for chunk in content:
>         yield alter_content(chunk)
> ```

process_exception

process_exception(request, exception)

request是一个HttpRequest对象。exception是一个被视图中的方法抛出来的Exception对象。

当一个视图抛出异常时，Django会调用process_exception()来处理。process_exception()应该返回一个None 或者一个HttpResponse对象。如果它返回一个HttpResponse对象，模型响应和响应中间件会被应用，响应结果会返回给浏览器。Otherwise, default exception handling kicks in.

再次提醒，在处理响应期间，中间件的执行顺序是倒序执行的，这包括process_exception。如果一个异常处理的中间件返回了一个响应，那这个中间件上面的中间件都将不会被调用。

__init__

大多数的中间件类都不需要一个初始化方法，因为中间件的类定义仅仅是为process_*提供一个占位符。如果你确实需要一个全局的状态那就可以通过__init__来加载。然后要铭记如下两个警告：

Django初始化你的中间件无需任何参数，因此不要定义一个有参数的__init__方法。 不像process_*每次请求到达都要调用__init__只会被调用一次，就是在Web服务启动的时候。

标记中间件不被使用

有时在运行时决定是否一个中间件需要被加载是很有用的。 在这种情况下，你的中间件中的 __init__方法可以抛出一个django.core.exceptions.MiddlewareNotUsed异常。Django会从中间件处理过程中移除这部分中间件，并且当DEBUG为True的时候在django.request记录器中记录调试信息。

```
1.8中的修改：

之前 MiddlewareNotUsed异常不会被记录。
```

指导准则

- 中间件的类不能是任何类的子类。
- 中间件可以存在与你Python路径中的任何位置。 Django所关心的只是被包含在MIDDLEWARE_CLASSES中的配置。
- 将Django’s available middleware作为例子随便看看。
- 如果你认为你写的中间件组建可能会对其他人有用，那就把它共享到社区！ 让我们知道它，我们会考虑把它添加到Django中。

中间件

这篇文档介绍了Django自带的所有中间件组件。 要查看关于如何使用它们以及如何编写自己的中间件，请见中间件使用指导。

可用的中间件

缓存中间件

class UpdateCacheMiddleware[source]

class FetchFromCacheMiddleware[source]

开启全站范围的缓存。 如果开启了这些缓存，任何一个由Django提供的页面将会被缓存，缓存时长是由你在CACHE_MIDDLEWARE_SECONDS配置中定义的。详见缓存文档。

"常用"的中间件

class CommonMiddleware[source]

给完美主义者增加一些便利条件：

- 禁止访问DISALLOWED_USER_AGENTS中设置的用户代理，这项配置应该是一个已编译的正则表达式对象的列表。
- 基于APPEND_SLASH和PREPEND_WWW的设置来重写URL。

如果APPEND_SLASH设为True并且一开始的的URL没有以斜线结尾，并且在URLconf中也没找到对应定义，这时形成一个一个斜线结尾新的URL。如果这个新的URL存在于URLconf，这时Django会重定向请求到这个新URL上，否则，一开始的URL按正常情况处理。

比如，foo.com/bar将会被重定向到foo.com/bar/，如果你没有为foo.com/bar定义有效的正则，但是为foo.com/bar/定义了有效的正则。

如果PREPEND_WWW设为True，前面缺少 "www."的url将会被重定向到相同但是以一个"www."开头的url。

两种选项都是为了规范化url。其中的哲学就是，任何一个url应该在一个地方仅存在一个。技术上来讲，url foo.com/bar 区别于foo.com/bar/ -- 搜索引擎索引会把这里分开处理 -- 因此，最佳实践就是规范化url。

- 基于 USE_ETAGS 设置来处理ETag。如果设置USE_ETAGS为True，Django会通过MD5-hashing处理页面的内容来为每一个页面请求计算Etag，并且如果合适的话，它将会发送携带Not Modified的响应。

CommonMiddleware.response_redirect_class

```
Django 1.8中新增
```

默认为HttpResponsePermanentRedirect。它继承了CommonMiddleware，并覆写了属性来自定义中间件发出的重定向。

class BrokenLinkEmailsMiddleware[source]

- 向MANAGERS发送死链提醒邮件（详见错误报告）。

GZip中间件

class GZipMiddleware[source]

> 警告
>
> 安全研究员最近发现，当压缩技术（包括GZipMiddleware）用于一个网站的时候，网站会受到一些可能的攻击。此外，这些方法可以用于破坏Django的CSRF保护。在你的站点使用GZipMiddleware之前，你应该先仔细考虑一下你的站点是否容易受到这些攻击。 如果你不确定是否会受到这些影响，应该避免使用 GZipMiddleware。详见the BREACH paper (PDF)和breachattack.com。

为支持GZip压缩的浏览器（一些现代的浏览器）压缩内容。

建议把这个中间件放到中间件配置列表的第一个，这样压缩响应内容的处理会到最后才 发生。

如果满足下面条件的话，内容不会被压缩：

- 消息体的长度小于200个字节。
- 响应已经设置了Content-Encoding协议头。
- 请求（浏览器）没有发送包含gzip的Accept-Encoding协议头。

你可以通过这个gzip_page()装饰器使用独立的GZip压缩。

带条件判断的GET中间件

class ConditionalGetMiddleware[source]

处理带有条件判断状态GET操作。 如果一个请求包含 ETag 或者Last-Modified协议头，并且请求包含If-None-Match或If-Modified-Since，这时响应会被 替换为HttpResponseNotModified。

另外，它会设置Date和Content-Length响应头。

本地中间件

class LocaleMiddleware[source]

基于请求中的数据开启语言选择。 它可以为每个用户进行定制。 详见国际化文档。

LocaleMiddleware.response_redirect_class

默认为HttpResponseRedirect。继承自LocaleMiddleware并覆写了属性来自定义中间件发出的重定向。

消息中间件

class MessageMiddleware[source]

开启基于cookie或会话的消息支持。详见消息文档。

安全中间件

> 警告
>
> 如果你的部署环境允许的话，让你的前端web服务器展示SecurityMiddleware提供的功能是个好主意。这样一来，如果有任何请求没有被Django处理（比如静态媒体或用户上传的文件），他们会拥有和向Django应用的请求相同的保护。

class SecurityMiddleware[source]

```
Django 1.8中新增
```

django.middleware.security.SecurityMiddleware为请求/响应循环提供了几种安全改进。每一种可以通过一个选项独立开启或关闭。

- SECURE_BROWSER_XSS_FILTER
- SECURE_CONTENT_TYPE_NOSNIFF
- SECURE_HSTS_INCLUDE_SUBDOMAINS
- SECURE_HSTS_SECONDS
- SECURE_REDIRECT_EXEMPT
- SECURE_SSL_HOST
- SECURE_SSL_REDIRECT

HTTP Strict Transport Security (HSTS)

对于那些应该只能通过HTTPS访问的站点，你可以通过设置HSTS协议头，通知现代的浏览器，拒绝用不安全的连接来连接你的域名。这会降低你受到SSL-stripping的中间人（MITM）攻击的风险。

如果你将SECURE_HSTS_SECONDS设置为一个非零值，SecurityMiddleware会在所有的HTTPS响应中设置这个协议头。

开启HSTS的时候，首先使用一个小的值来测试它是个好主意，例如，让SECURE_HSTS_SECONDS = 3600为一个小时。每当浏览器在你的站点看到HSTS协议头，都会在提供的时间段内绝对使用不安全（HTTP）的方式连接到你的域名。一旦你确认你站点上的所有东西都以安全的方式提供（例如，HSTS并不会干扰任何事情），建议你增加这个值，这样不常访问你站点的游客也会被保护（比如，一般设置为31536000秒，一年）。

另外，如果你将 SECURE_HSTS_INCLUDE_SUBDOMAINS设置为True,，SecurityMiddleware会将includeSubDomains标签添加到Strict-Transport-Security协议头中。强烈推荐这样做（假设所有子域完全使用HTTPS），否则你的站点仍旧有可能由于子域的不安全连接而受到攻击。

> 警告
>
> HSTS策略在你的整个域中都被应用，不仅仅是你所设置协议头的响应中的url。所以，如果你的整个域都设置为HTTPS only，你应该只使用HSTS策略。
>
> 适当遵循HSTS协议头的浏览器，会通过显示警告的方式，拒绝让用户连接到证书过期的、自行签署的、或者其他SSL证书无效的站点。如果你使用了HSTS，确保你的证书处于一直有效的状态！
>
> 注意
>
> 如果你的站点部署在负载均衡器或者反向代理之后，并且Strict-Transport-Security协议头没有添加到你的响应中，原因是Django有可能意识不到这是一个安全连接。你可能需要设置SECURE_PROXY_SSL_HEADER。

X-Content-Type-Options: nosniff

一些浏览器会尝试猜测他们所得内容的类型，而不是读取Content-Type协议头。虽然这样有助于配置不当的服务器正常显示内容，但也会导致安全问题。

如果你的站点允许用户上传文件，一些恶意的用户可能会上传一个精心构造的文件，当你觉得它无害的时候，文件会被浏览器解释成HTML或者Javascript。

欲知更多有关这个协议头和浏览器如何处理它的内容，你可以在IE安全博客中读到它。

要防止浏览器猜测内容类型，并且强制它一直使用 Content-Type协议头中提供的类型，你可以传递X-Content-Type-Options: nosniff协议头。SecurityMiddleware将会对所有响应这样做，如果SECURE_CONTENT_TYPE_NOSNIFF 设置为True。

注意在大多数Django不涉及处理上传文件的部署环境中，这个设置不会有任何帮助。例如，如果你的MEDIA_URL被前端web服务器直接处理（例如nginx和Apache），你可能想要在那里设置这个协议头。而在另一方面，如果你使用Django执行为了下载文件而请求授权之类的事情，并且你不能使用你的web服务器设置协议头，这个设置会很有用。

X-XSS-Protection: 1; mode=block

一些浏览器能够屏蔽掉出现XSS攻击的内容。通过寻找页面中GET或者POST参数中的JavaScript内容来实现。如果JavaScript在服务器的响应中被重放，页面就会停止渲染，并展示一个错误页来取代。

X-XSS-Protection协议头用来控制XSS过滤器的操作。

要在浏览器中启用XSS过滤器，并且强制它一直屏蔽可疑的XSS攻击，你可以在协议头中传递X-XSS-Protection: 1; mode=block。 如果SECURE_BROWSER_XSS_FILTER设置为True，SecurityMiddleware会在所有响应中这样做。

> 警告
>
> 浏览器的XSS过滤器是一个十分有效的手段，但是不要过度依赖它。它并不能检测到所有的XSS攻击，也不是所有浏览器都支持这一协议头。确保你校验和过滤了所有的输入来防止XSS攻击。

SSL重定向

如果你同时提供HTTP和HTTPS连接，大多数用户会默认使用不安全的（HTTP）链接。为了更高的安全性，你应该讲所有HTTP连接重定向到HTTP连接。

如果你将SECURE_SSL_REDIRECT设置为True，SecurityMiddleware会将HTTP链接永久地（HTTP 301，permanently）重定向到HTTPS连接。

> 注意
>
> 由于性能因素，最好在Django外面执行这些重定向，在nginx这种前端负载均衡器或者反向代理服务器中执行。SECURE_SSL_REDIRECT专门为这种部署情况而设计，当这不可选择的时候。

如果SECURE_SSL_HOST设置有一个值，所有重定向都会发到值中的主机，而不是原始的请求主机。

如果你站点上的一些页面应该以HTTP方式提供，并且不需要重定向到HTTPS，你可以SECURE_REDIRECT_EXEMPT设置中列出匹配那些url的正则表达式。

> 注意
>
> 如果你在负载均衡器或者反向代理服务器后面部署应用，而且Django不能辨别出什么时候一个请求是安全的，你可能需要设置SECURE_PROXY_SSL_HEADER。

会话中间件

class SessionMiddleware[source]

开启会话支持。详见会话文档。

站点中间件

class CurrentSiteMiddleware[source]

```
Django 1.7中新增
```

向每个接收到的HttpRequest对象添加一个site属性，表示当前的站点。详见站点文档。

认证中间件

class AuthenticationMiddleware[source]

向每个接收到的HttpRequest对象添加user属性，表示当前登录的用户。详见web请求中的认证。

class RemoteUserMiddleware[source]

使用web服务器提供认证的中间件。详见使用REMOTE_USER进行认证。

class SessionAuthenticationMiddleware[source]

```
Django 1.7中新增
```

当用户修改密码的时候使用户的会话失效。详见密码更改时的会话失效。在MIDDLEWARE_CLASSES中，这个中间件必须出现在django.contrib.auth.middleware.AuthenticationMiddleware之后。

CSRF保护中间件

class CsrfViewMiddleware[source]

添加跨站点请求伪造的保护，通过向POST表单添加一个隐藏的表单字段，并检查请求中是否有正确的值。详见CSRF保护文档。

X-Frame-Options中间件

class XFrameOptionsMiddleware[source]

通过X-Frame-Options协议头进行简单的点击劫持保护。

中间件的排序

下面是一些关于Django中间件排序的提示。

UpdateCacheMiddleware

放在修改大量协议头的中间件(SessionMiddleware, GZipMiddleware, LocaleMiddleware)之前。

GZipMiddleware

放在任何可能修改或使用响应消息体的中间件之前。

放在UpdateCacheMiddleware之后：会修改大量的协议头。

ConditionalGetMiddleware

放在CommonMiddleware之前：当USE_ETAGS = True时会使用它的Etag 协议头。

SessionMiddleware

放在UpdateCacheMiddleware之后：会修改 大量协议头。

LocaleMiddleware

放在SessionMiddleware（由于使用会话数据）和 CacheMiddleware（由于要修改大量协议头）之后的最上面。

CommonMiddleware

放在任何可能修改相应的中间件之前（因为它会生成ETags）。

在GZipMiddleware之后，不会在压缩后的内容上再去生成ETag。

尽可能放在靠上面的位置，因为APPEND_SLASH或者PREPEND_WWW设置为True时会被重定向。

CsrfViewMiddleware

放在任何假设CSRF攻击被处理的视图中间件之前。

AuthenticationMiddleware

放在SessionMiddleware之后：因为它使用会话存储。

MessageMiddleware

放在SessionMiddleware之后：会使用基于会话的存储。

FetchFromCacheMiddleware

放在任何修改大量协议头的中间件之后：协议头被用来从缓存的哈希表中获取值。

FlatpageFallbackMiddleware

应该放在最底下，因为他是中间件中的底牌。

RedirectFallbackMiddleware

应该放在最底下，因为他是中间件中的底牌。

模板层

模板层提供了设计友好的语法来展示信息给用户。 了解其语法可以让设计师知道如何使用，让程序员知道如何扩展：

基础

模版

作为Web 框架，Django 需要一种很便利的方法以动态地生成HTML。最常见的做法是使用模板。模板包含所需HTML 输出的静态部分，以及一些特殊的语法，描述如何将动态内容插入。创建HTML 页面模板的一个示例，参见教程第3部分。

Django 项目可以配置一个或多个模板引擎（甚至是零，如果你不需要使用模板）。Django 的模板系统自带内建的后台 —— 称为Django 模板语言（DTL），以及另外一种流行的Jinja2。其他的模板语言的后端，可查找第三方库。

Django 为加载和渲染模板定义了一套标准的API，与具体的后台无关。加载包括根据给定的标识找到模板然后预处理，通常会将它编译好放在内存中。渲染表示使用Context 数据对模板插值并返回生成的字符串。

Django 模板语言 是Django 原生的模板系统。直到Django 1.8，这是唯一可用的内置选项。尽管，它闭门造车，并且偏重某些方面，但是它仍然是一个优秀的模版库。如果没有特别紧急的理由选择另外一种后台，你应该使用DTL，特别是你编写可插拔的应用并打算发布其模板的时候。Django 中包含模板的标准应用，例如*django.contrib.admin*，都使用DTL。

又由于历史遗留原因,通用支持的模板引擎和Django实现的模板语言都在 `django.template` 命名空间中.

模板引擎的支持

New in Django 1.8:

Django1.8 中增加了对多种模板引擎的支持和 `TEMPLATES` 设置。

配置

模板引擎通过 `TEMPLATES` 设置来配置。它是一个设置选项列表，与引擎一一对应。默认的值为空。由 `startproject` 命令生成的 `settings.py` 定义了一些有用的值：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTempla
tes',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

`BACKEND` 是一个指向实现了Django模板后端API的模板引擎类的带点的Python路径。内置的后端有 `django.template.backends.django.DjangoTemplates` 和 `django.template.backends.jinja2.Jinja2` .

由于绝大多数引擎都是从文件加载模板的，所以每种模板引擎都包含两项通用设置：

- `DIRS` 定义了一个目录列表，模板引擎按列表顺序搜索这些目录以查找模板源文件。
- `APP_DIRS` 告诉模板引擎是否应该进入每个已安装的应用中查找模板。每种模板引擎后端都定义了一个惯用的名称作为应用内部存放模板的子目录名称。（译者注：例如django为它自己的模板引擎指定的是 'templates' ，为jinja2指定的名字是'jinja2'）

特别的是，django允许你有多个模板引擎后台实例，且每个实例有不同的配置选项。在这种情况下你必须为每个配置指定一个唯一的 `NAME` .

`OPTIONS` 中包含了具体的backend设置

用法

`django.template.loader` 定义了两个函数以加载模板。

`get_template` (*template_name[, dirs][, using]*)[source]

该函数使用给定的名称加载模板并返回一个 `Template` 对象.

眞正的返回值类型取决于那个用来加载模版的后台引擎。每个后台都有各自的 `Template` 类。

`get_template()` 尝试获取每个模板直到有一个成功满足。如果模板不能成功找到，将会抛出 `TemplateDoesNotExist` . 如果能够找到模板但是包含非法值，将会抛出 `TemplateSyntaxError` .

模板的查找和加载机制取决于每种后台引擎和配置

如果你想使用指定的模板引擎进行查找,请将模板引擎的 `NAME` 赋给 get_template 的 `using` 参数

Changed in Django 1.7:

添加了 `dirs` 参数。

自1.8版起已弃用： `dirs` 参数被移除。

Changed in Django 1.8:

添加了 参数 `using` 。

Changed in Django 1.8:

`get_template()` 返回了一个基于后端引擎的 `Template` 而不是 `django.template.Template` .这个类。

`select_template` (*template_name_list[, dirs][, using]*)[source]

`select_template()` 和 `get_template()` 很相似, 只不过它用了一个模板名称的列表作为参数。按顺序搜索模板名称列表内的模板并返回第一个存在的模板。

Changed in Django 1.7:

增加 `dirs` 参数。

自1.8版起已弃用：废弃 `dirs` 参数。

Changed in Django 1.8:

已添加 `using` 参数。

Changed in Django 1.8:

`select_template()` 返回与后端相关的 `Template` ，而不是 `django.template.Template` 。

如果导入模板失败， `django.template` 中定义的两个异常，有可能会被抛出：

exception `TemplateDoesNotExist`

在找不到该模板时，抛出该错误。

exception `TemplateSyntaxError`

当模板内出现错误时，将被抛出

由 `get_template()` 和 `select_template()` 返回的 `Template` 对象必须要有一个 `render()` 方法，协议如下：

`Template.``render` (*context=None*, *request=None*)

通过给定的 context 对该模板进行渲染。

如果提供了 `context` ，那么它必须是一个 `dict` 对象. 如果没有提供，引擎将是用空 context 对模板进行渲染。

如果要提供 `request` 参数 ，必须使用 `HttpRequest` 对象. 之后模板引擎会使它连同CSRF token一起在模板中可用.具体如何实现由相应模板后端决定.

关于搜索算法的例子。该例子下 `TEMPLATES` 的配置是:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTempla
tes',
        'DIRS': [
            '/home/html/example.com',
            '/home/html/default',
        ],
    },
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'DIRS': [
            '/home/html/jinja2',
        ],
    },
]
```

如果你调用函数 `get_template('story_detail.html')` , Django按以下顺序查找story_detail.html：

- `/home/html/example.com/story_detail.html` （ `'django'` engine）
- `/home/html/default/story_detail.html` （ `'django'` engine）
- `/home/html/jinja2/story_detail.html` （ `'jinja2'` 引擎）

如果你调用函数 `select_template(['story_253_detail.html', 'story_detail.html'])` , Django按一下顺序查找：

- `/home/html/example.com/story_253_detail.html` （ `'django'` engine）
- `/home/html/default/story_253_detail.html` （ `'django'` 引擎）
- `/home/html/jinja2/story_253_detail.html` （ `'jinja2'` 引擎）
- `/home/html/example.com/story_detail.html` （ `'django'` engine）
- `/home/html/default/story_detail.html` （ `'django'` engine）
- `/home/html/jinja2/story_detail.html` （ `'jinja2'` 引擎）

当Django发现模板存在后便停止搜寻。

提示

你可以通过 `select_template()` 来实现更为灵活的模板加载。例如，如果您撰写了新闻报道并希望某些报道有自定义模板，请使用 select_template（['story_%s_detail.html' % t2 > story.id，'st。这允许你为每个故事使用一个个性化的模板,同时有一个后背模板给没有个性化模板的故事使用.

如果可能（其实这会更好）—将模版文件，放在包含模版的目录的子目录下。基本思路是，使得每个APP的的模版子目录下都有一个子目录来唯一对应这个APP。

这样做可以增强你的APP可用性。将所有的模版放在根模版目录下会引发混淆。

要在一个子目录内加载模板，像这样使用斜线就好了：

```
get_template('news/story_detail.html')
```

作为上述使用相同的 `模板` 选项，这将会尝试加载下列模板：

- `/home/html/example.com/news/story_detail.html` （ `'django'` engine）
- `/home/html/default/news/story_detail.html` （ `'django'` engine）
- `/home/html/jinja2/news/story_detail.html` （ `'jinja2'` 引擎）

另外，为了减少加载模板、渲染模板等重复工作，django提供了处理这些工作的快捷函数。

`render_to_string` (*template_name[, context][, context_instance][, request][, using]*)[source]

`render_to_string()` 会像 `get_template()` 一样加载模板并立即调用 `render()` 方法。 它需要以下参数。

`template_name`

The name of the template to load and render. If it's a list of template names, Django uses `select_template()` instead of `get_template()` to find the template.

`context`

要用作模板的上下文进行渲染的 `dict` 。

Changed in Django 1.8:

用于调用 `dictionary` 的 `context` 参数。该名称在Django 1.8中已弃用，将在Django 2.0中删除。

`context` 现在是可选的。如果没有提供空的上下文将被使用。

`context_instance`

一个 `Context` t实例，或者是子类的实例（例如， `RequestContext` 的实例），可以像template的context一样使用它。

自1.8版起已弃用： `context_instance` 已经被废弃。使用 `context` 和如果需要 `request` 。

`request`

`HttpRequest` 是可选的，并且在整个模版渲染期都是可用的。

New in Django 1.8:

已添加 `request` 参数。

同时请看 `render()` 和 `render_to_response()` 快捷方式，它们调用 `render_to_string()` ，并将渲染结果放入到一个合适的 `HttpResponse` 中用以从view中返回

最后，您可以直接使用配置的引擎：

`engines`

模板引擎可在 `django.template.engines` 中使用：

```
from django.template import engines

django_engine = engines['django']
template = django_engine.from_string("Hello {{ name }}!")
```

在此示例中，查找键 - `'django'` 是引擎的 `NAME` 。

内置(模板)后端

class `DjangoTemplates` [source]

设置 `BACKEND` 为 `'django.template.backends.django.DjangoTemplates'` 来配置Django模板引擎。

当 `APP_DIRS` 为 `True` 时, `DjangoTemplates` 引擎会在已安装应用的 `templates` 子目录中查找模板文件。 这个通用名称是保持向后兼容的。

`DjangoTemplates` 引擎 `OPTIONS` 配置项中接受以下参数:

- `'allowed_include_roots'` : 这是一个字符串列表，表示这些字符串允许出现在 `{% ssi %}` 中，作为被允许使用的模板标签。这是一个安全的措施，使模板作者不能访问他们不应该访问的文件。

  例如, 如果 `'allowed_include_roots'` 是 `['/home/html', '/var/www']` ,那么 `{% ssi /home/html/foo.txt %}` 就会生效, 但是 `{% ssi /etc/passwd %}` 就无效。

它默认是个空的 list.

自1.8版起已弃用： `allowed_include_roots` 已弃用，因为已弃用{%ssi%}标记。

- `'context_processors'`：是一个包含以"."为分隔符的python调用路径的列表，在一个template被request渲染时，它可以被调用以产生context的数据。这些可调用要求一个请求对象作为其参数，并返回要合并到上下文中的项目的 `dict` 。

  它默认为空列表。

  有关详细信息，请参阅 `RequestContext` 。

- `'debug'` ：打开/关闭模板调试模式的布尔值。如果它 `True` ，那么奇怪的错误页面将显示模板渲染期间引发的任何异常的详细报告。此报告包含模板的相关代码段，并突出显示相应的行。

  它默认和setting中的 `DEBUG` 有相同的值。

- `'loaders'` ：模板加载器类的虚拟Python路径列表。每个 `Loader` 类知道如何从特定源导入模板。你可以选择使用字符串元组来代替字符串。元组的第一项是 `Loader` 类名，接下来的项在初始化期间会被传递给 `Loader` 。

  默认值取决于 `DIRS` 和 `APP_DIRS` 的值。

  有关详细信息，请参见*Loader types*。

- `'string_if_invalid'` ：作为字符串的输出，模板系统应该用于无效（例如拼写错误的）变量。

  它默认为空字符串。

  有关详细信息，请参见*How invalid variables are handled*。

- `'file_charset'` ：用于读取磁盘上的模板文件的字符集。

  它默认为 `FILE_CHARSET` 的值。

class `Jinja2` [source]

需要安装 Jinja2：

```
$ pip install Jinja2
```

设置 `BACKEND` 为 `'django.template.backends.jinja2.Jinja2'` 时，则启用 Jinja2 引擎。

当 `APP_DIRS` 为 `True` 时， `Jinja2` 引擎则在已安装应用的名为 `jinja2` 的子目录中查找模板文件。

最重要的配置项是 `OPTIONS` 中的 `'environment'` 项。它应该是一个带点的Python路径，调用它则可返回 Jinja2 的环境变量。它的默认值是 `'jinja2.` Environment'。Django调用该可调用项并传递其他选项作为关键字参数。此外，Django添加了与Jinja2不同的默认值，用于几个选项：

- `'autoescape'` ： `True`
- `'loader'` ：为 `DIRS` 和 `APP_DIRS`
- `'auto_reload'` ： `settings.` 调试
- `'undefined'` ： `DebugUndefined if settings。` DEBUG else 未定义

有意将默认配置保持为最小。 `Jinja2` 后端不会创建Django风格的环境。它不知道Django上下文处理器，过滤器和标签。为了使用Django特定的API，您必须将它们配置到环境中。

例如，你可以创建一个包含以下内容的 `myproject/jinja2.py` ：

```
from __future__ import absolute_import  # Python 2 only

from django.contrib.staticfiles.storage import staticfiles_stora
ge
from django.core.urlresolvers import reverse

from jinja2 import Environment

def environment(**options):
    env = Environment(**options)
    env.globals.update({
        'static': staticfiles_storage.url,
        'url': reverse,
    })
    return env
```

并将 `'environment'` 选项设置为 `'myproject.jinja2.environment'` 。

之后你可以用下面的内容构建Jinja2模板：

```
<img src="{{ static('path/to/company-logo.png') }}" alt="Company
 Logo">

<a href="{{ url('admin:index') }}">Administration</a>
```

标签和过滤器的概念尽管在Django模板语言和Jinja2中都存在，但使用起来却不相同。因为Jinja2支持传递参数给模版中的可调用对象，在Django中许多需要模版标记和过滤的特性，在Jinja2可以通过简单的调用函数来得到，比如上面所显示的例子。Jinja2的全局命名空间移除了模版上下文处理器的需求。Django模版语言没有等同于Jinja2 tests的东西。

定制后端

为了使用其他的模板系统，这里将介绍如何实现一个自定义模板后台。模板后台都是继承自 `django.template.backends.base.BaseEngine` 类的。它必须实现 `get_template()` 方法，也可实现可选函数 `from_string()` 。下面是一个虚构的 `foobar` 模板库的示例：

```
from django.template import TemplateDoesNotExist, TemplateSyntax
Error
from django.template.backends.base import BaseEngine
from django.template.backends.utils import csrf_input_lazy, csrf
_token_lazy

import foobar

class FooBar(BaseEngine):

    # Name of the subdirectory containing the templates for this
 engine
    # inside an installed application.
    app_dirname = 'foobar'

    def __init__(self, params):
        params = params.copy()
        options = params.pop('OPTIONS').copy()
        super(FooBar, self).__init__(params)

        self.engine = foobar.Engine(**options)

    def from_string(self, template_code):
        try:
          return Template(self.engine.from_string(template_code)
)
        except foobar.TemplateCompilationFailed as exc:
            raise TemplateSyntaxError(exc.args)

    def get_template(self, template_name):
        try:
            return Template(self.engine.get_template(template_na
me))
        except foobar.TemplateNotFound as exc:
            raise TemplateDoesNotExist(exc.args)
        except foobar.TemplateCompilationFailed as exc:
            raise TemplateSyntaxError(exc.args)

class Template(object):

    def __init__(self, template):
        self.template = template

    def render(self, context=None, request=None):
        if context is None:
            context = {}
        if request is not None:
            context['request'] = request
            context['csrf_input'] = csrf_input_lazy(request)
            context['csrf_token'] = csrf_token_lazy(request)
        return self.template.render(context)
```

有关详细信息，请参阅DEP 182。

Django模板语言

语法

关于本节

以下是关于Django模板语法的概览。更多细节请见 模板语言语法参考.

Django模板是一个简单的文本文档，或用Django模板语言标记的一个Python字符串。 某些结构是被模板引擎解释和识别的。主要的有变量和标签。

模板是由context来进行渲染的。渲染的过程是用在context中找到的值来替换模板中相应的变量，并执行相关tags。其他的一切都原样输出。

Django模板语言的语法包括四个结构。

变量

变量的值是来自context中的输出, 这类似于字典对象的keys到values的映射关系。

变量是被 `{{` 和 `}}` 括起来的部分，例如：

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

如果使用一个 context包含 `{'first_name': 'John', 'last_name': 'Doe'}`，这个模板渲染后的情况将是:

```
My first name is John. My last name is Doe.
```

字典查询，属性查询和列表索引查找都是通过一个点符号来实现：

```
{{ my_dict.key }}
{{ my_object.attribute }}
{{ my_list.0 }}
```

如果一个变量被解析为一个可调用的，模板系统会调用它不带任何参数，并使用调用它的结果来代替这个可调用对象本身。

标签

标签在渲染的过程中提供任意的逻辑。

这个定义是刻意模糊的。例如，一个标签可以输出内容，作为控制结构，例如“if”语句或“for”循环从数据库中提取内容，甚至可以访问其他的模板标签。

Tags是由 `{%` 和 `%}` 来定义的，例如：

```
{% csrf_token %}
```

大部分标签都接受参数

```
{%< cycle 'odd' 'even' %}
```

部分标签要求使用起始和闭合标签：

```
{%< if user.is_authenticated %}Hello, {{ user.username }}.{%< en
dif %}
```

*Django*内置标签参考文档 和 编写定制化标签指引都可以参阅。

过滤器

过滤器会更改变量或标签参数的值。

看上去像这样：

```
{{ django|title }}
```

例如在 `{'django': 'the web framework for perfectionists with deadlines'}` 这个context中，django变量的值都是小写，经title过滤器渲染后则变成：

```
The Web Framework For Perfectionists With Deadlines
```

有些过滤器看起来更像参数：

```
{{ my_date|date:"Y-m-d" }}
```

具体可以查看 内置过滤器参考和 开发自定义过滤器指南这两篇文档.

注释

注释看起来像这样:{# this won't be rendered #}

A `{% comment %}` 标签提供多行注释。

组件

关于这个部分

本文只是对Django模板语言API作一个简单概述. 具体请参阅 *API*参考指引.

引擎

`django.template.Engine` 封装Django模板系统的一个实例。其主要原因是直接实例化一个 `引擎` 在Django项目之外进行使用。

`django.template.backends.django.DjangoTemplates` 是 `django.template.E` 的简化版,他们都是指向Django模板后端的API。

模板

`django.template.Template` 表示一个已编译的模板。模板可以通过 `Engine.get_template()` 或 `Engine.from_string()` 来获取。

同样地, `django.template.backends.django.Template` 是 `django.template.Template` 的一个简化版，他们指向共同的模板API.

上下文

`django.template.Context` 除了持有context数据以外，还持有一些元数据. 它被传递到 `Template.render()` 用于渲染一个模板.

`django.template.RequestContext` 是 `Context` 的一个子类，它存储当前的 `HttpRequest` 并且运行了模板的context处理器.

一般的API不具备这样的概念。大多数情况下，context数据是在一个普通的 `字典` 里传递，而当前的 `HttpRequest` 如果有需要的话是另外传递的。

加载器

模板加载器负责定位模板，加载它们，并返回 `模板` 对象.

Django提供几个内置的模板加载器并且支持自定义的模板加载器.

上下文处理器

Context处理器是这样的函数：接收当前的 `HttpRequest` 作为参数，并返回一个 `字典` ，该字典中包含了将要添加到渲染的context中的数据。

它们的主要用途是添加所有的模板context共享的公共数据，而不需要在每个视图中重复代码。

Django提供了很多 内置的*context*处理器 . 实现自定义context处理器很简单，只要定义一个函数。

面向设计师

Django模版语言

本文将介绍Django模版系统的语法。如果您需要更多该系统如何工作的技术细节，以及希望扩展它，请浏览 *The Django template language: for Python programmers*.

Django模版语言的设计致力于在性能和简单上取得平衡。 它的设计使习惯于使用HTML的人也能够自如应对。如果您有过使用其他模版语言的经验，像是 Smarty 或者 Jinja2, 那么您将对Django的模版语言感到一见如故。

理念

如果您有过编程背景，或者您使用过一些在HTML中直接混入程序代码的语言，那么现在您需要记住，Django的模版系统并不是简单的将Python嵌入到HTML中。 设计决定了：模版系统致力于表达外观，而不是程序逻辑。

Django的模版系统提供了和一些程序结构功能类似的标签——用于布尔判断的 `if` 标签, 用于循环的 `for` 标签等等。——但是这些都不是简单的作为Python代码那样来执行的，并且，模版系统也不会随意执行Python表达式。只有下面列表中的标签、过滤器和语法才是默认就被支持的。 （但是您也可以根据需要添加 您自己的扩展 到模版语言中）。

模版

模版是纯文本文件。它可以产生任何基于文本的的格式（HTML，XML，CSV等等）。

模版包括在使用时会被值替换掉的 变量，和控制模版逻辑的 标签。

下面是一个小模版，它说明了一些基本的元素。后面的文档中会解释每个元素。

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

理念

为什么要使用基于文本的模版，而不是基于XML的（比如Zope的TAL）呢？我们希望Django的模版语言可以用在更多的地方，而不仅仅是XML/HTML模版。在线上世界，我们在email、Javascript和CSV中使用它。你可以在任何基于文本的格式中使用这个模版语言。

还有，让人类编辑HTML简直是施虐狂的做法！

变量

变量看起来就像是这样： `{{ variable }}`. 当模版引擎遇到一个变量，它将计算这个变量，然后用结果替换掉它本身。变量的命名包括任何字母数字以及下划线 (`"_"`)的组合。点(`"."`) 也在会变量部分中出现，不过它有特殊的含义，我们将在后面说明。重要的是， 你不能在变量名称中使用空格和标点符号。

使用点 (`.`) 来访问变量的属性。

幕后

从技术上来说，当模版系统遇到点，它将以这样的顺序查询：

- 字典查询（Dictionary lookup）
- 属性或方法查询（Attribute or method lookup）
- 数字索引查询（Numeric index lookup）

如果计算结果的值是可调用的，它将被无参数的调用。调用的结果将成为模版的值。

这个查询顺序，会在优先于字典查询的对象上造成意想不到的行为。例如，思考下面的代码片段，它尝试循环 `collections.defaultdict`：

```
{% for k, v in defaultdict.iteritems %}
    Do something with k and v here...
{% endfor %}
```

因为字典查询首先发生，行为奏效了并且提供了一个默认值，而不是使用我们期望的 `.iteritems()` 方法。在这种情况下，考虑首先转换成字典。

在前文的例子中， `{{ section.title }}` 将被替换为 `section` 对象的 `title` 属性。

如果你使用的变量不存在， 模版系统将插入 `string_if_invalid` 选项的值， 它被默认设置为 `''` (空字符串) 。

注意模版表达式中的“bar”， 比如 `{{ foo.bar }}` 将被逐字直译为一个字符串，而不是使用变量“bar”的值，如果这样一个变量在模版上下文中存在的话。

过滤器

您可以通过使用 过滤器来改变变量的显示。

过滤器看起来是这样的：`{{ name|lower }}`。这将在变量 `{{ name }}` 被过滤器 `lower` 过滤后再显示它的值，该过滤器将文本转换成小写。使用管道符号(`|`)来应用过滤器。

过滤器能够被“串联”。一个过滤器的输出将被应用到下一个。`{{ text|escape|linebreaks }}` 就是一个常用的过滤器链，它编码文本内容，然后把行打破转成 `<p>` 标签。

一些过滤器带有参数。过滤器的参数看起来像是这样：`{{ bio|truncatewords:30 }}`。这将显示 `bio` 变量的前30个词。

过滤器参数包含空格的话，必须被引号包起来；例如，连接一个有逗号和空格的列表，你需要使用 `{{ list|join:", " }}`。

Django提供了大约六十个内置的模版过滤器。你可以在 内置过滤器参考手册中阅读全部关于它们的信息。为了体验一下它们的作用，这里有一些常用的模版过滤器：

`default`

如果一个变量是false或者为空，使用给定的默认值。否则，使用变量的值。例如：

```
{{ value|default:"nothing" }}
```

如果 `value` 没有被提供，或者为空， 上面的例子将显示“`nothing`”。

`length`

返回值的长度。它对字符串和列表都起作用。例如：

```
{{ value|length }}
```

如果 `value` 是 `['a', 'b', 'c', 'd']`，那么输出是 `4`。

`filesizeformat`

将值格式化为一个 “人类可读的” 文件尺寸 （例如 `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, 等等）。例如：

```
{{ value|filesizeformat }}
```

如果 `value` 是 123456789，输出将会是 `117.7 MB`。

再说一下，这仅仅是一些例子；查看 内置过滤器参考手册 来获取完整的列表。

您也可以创建自己的自定义模版过滤器；参考 自定义模版标签和过滤器 。

更多

Django’s admin interface can include a complete reference of all template tags and filters available for a given site. See *The Django admin documentation generator*.

标签

标签看起来像是这样的： `{% tag %}` 。标签比变量更加复杂：一些在输出中创建文本，一些通过循环或逻辑来控制流程，一些加载其后的变量将使用到的额外信息到模版中。

一些标签需要开始和结束标签 （例如 `{% tag %} ...` 标签 内容 ... {% endtag %}）。

Django自带了大约24个内置的模版标签。你可以在 内置标签参考手册中阅读全部关于它们的内容。为了体验一下它们的作用，这里有一些常用的标签：

`for`

循环数组中的每个元素。例如，显示 `athlete_list` 中提供的运动员列表：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

`if` , `elif` , and `else`

计算一个变量，并且当变量是“true”是，显示块中的内容：

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

在上面的例子中，如果 `athlete_list` 不是空的，运动员的数量将显示为 `{{ athlete_list|length }}` 的输出。另一方面， 如果 `athlete_in_locker_room_list` 不为空， 将显示 “Athletes should be out...” 这个消息。如果两个列表都是空的，将显示 “No athletes.” 。

您也可以在 `if` 标签中使用过滤器和多种运算符：

```
{% if athlete_list|length > 1 %}
   Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
   Athlete: {{ athlete_list.0.name }}
{% endif %}
```

当上面的例子工作时，需要注意，大多数模版过滤器返回字符串，所以使用过滤器做数学的比较通常都不会像您期望的那样工作。 `length` 是一个例外。

`block` and `extends`

Set up template inheritance (see below), a powerful way of cutting down on “boilerplate” in templates.

再说一下，上面的仅仅是整个列表的一部分；查看 内置标签参考手册 来获取完整的列表。

您也可以创建您自己的自定义模版标签；参考 自定义模版标签和过滤器 。

更多

Django’s admin interface can include a complete reference of all template tags and filters available for a given site. See *The Django admin documentation generator*.

注释

要注释模版中一行的部分内容，使用注释语法 `{# #}` .

例如，这个模版将被渲染为 `'hello'` ：

```
{# greeting #}hello
```

注释可以包含任何模版代码，有效的或者无效的都可以。例如：

```
{# {% if foo %}bar{% else %} #}
```

这个语法只能被用于单行注释（在 `{#` 和 `#}` 分隔符中，不允许有新行）。如果你需要注释掉模版中的多行内容，请查看 `comment` 标签。

模版继承

Django模版引擎中最强大也是最复杂的部分就是模版继承了。模版继承可以让您创建一个基本的“骨架”模版，它包含您站点中的全部元素，并且可以定义能够被子模版覆盖的 **blocks** 。

通过从下面这个例子开始，可以容易的理解模版继承：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{%/span> endblock %}<
/title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

这个模版，我们把它叫作 `base.html` ，它定义了一个可以用于两列排版页面的简单HTML骨架。“子模版”的工作是用它们的内容填充空的blocks。

在这个例子中， `block` 标签定义了三个可以被子模版内容填充的block。 `block` 告诉模版引擎： 子模版可能会覆盖掉模版中的这些位置。

子模版可能看起来是这样的：

```
{% extends "base.html" %}/span>

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

`extends` 标签是这里的关键。它告诉模版引擎，这个模版“继承”了另一个模版。当模版系统处理这个模版时，首先，它将定位父模版——在此例中，就是“base.html”。

那时，模版引擎将注意到 `base.html` 中的三个 `block` 标签，并用子模版中的内容来替换这些block。根据 `blog_entries` 的值，输出可能看起来是这样的：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

请注意，子模版并没有定义 `sidebar` block，所以系统使用了父模版中的值。父模版的 `{% block %}` 标签中的内容总是被用作备选内容（fallback）。

您可以根据需要使用多级继承。使用继承的一个常用方式是类似下面的三级结构：

- 创建一个 `base.html` 模版来控制您整个站点的主要视觉和体验。
- 为您的站点的每一个“部分”创建一个 `base_SECTIONNAME.html` 模版。例如， `base_news.html` , `base_sports.html` 。这些模版都继承自 `base.html` ，并且包含了每部分特有的样式和设计。
- 为每一种页面类型创建独立的模版，例如新闻内容或者博客文章。这些模版继承了有关的部分模版（section template）。

这种方式使代码得到最大程度的复用，并且使得添加内容到共享的内容区域更加简单，例如，部分范围内的导航。

这里是使用继承的一些提示：

- 如果你在模版中使用 `{% extends %}` 标签，它必须是模版中的第一个标签。其他的任何情况下，模版继承都将无法工作。
- 在base模版中设置越多的 `{% block %}` 标签越好。请记住，子模版不必定义全部父模版中的blocks，所以，你可以在大多数blocks中填充合理的默认内容，然后，只定义你需要的那一个。多一点钩子总比少一点好。
- 如果你发现你自己在大量的模版中复制内容，那可能意味着你应该把内容移动到父模版中的一个 `{% block %}` 中。
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.
- 为了更好的可读性，你也可以给你的 `{% endblock %}` 标签一个 名字 。例如：

  ```
  {% block content %}
  ...
  {% endblock content %}
  ```

  在大型模版中，这个方法帮你清楚的看到哪一个 `{% block %}` 标签被关闭了。

最后，请注意您并不能在一个模版中定义多个相同名字的 `block` 标签。这个限制的存在是因为block标签的作用是“双向”的。这个意思是，block标签不仅提供了一个坑去填，它还在 父模版中定义了填坑的内容。如果在一个模版中有两个名字一样的 `block` 标签，模版的父模版将不知道使用哪个block的内容。

自动HTML转义

当从模版中生成HTML时，总会有这样一个风险：值可能会包含影响HTML最终呈现的字符。例如，思考这个模版片段：

```
Hello, {{ name }}
```

首先，它看起来像是一个无害的方式来显示用户的名字，但是设想一下，如果用户像下面这样输入他的名字，会发生什么：

```
<script>alert('hello')</script>
```

使用这个名字值，模版将会被渲染成这样：

```
Hello, <script>alert('hello')</script>
```

...这意味着，浏览器将会弹出一个Javascript警示框！

类似的，如果名字包含一个 `'<'` 符号（比如下面这样），会发生什么呢？

```
<b>username
```

这将会导致模版呗渲染成这样：

```
Hello, <b>username
```

...进而这将导致网页的剩余部分都被加粗！

显然，用户提交的数据都被应该被盲目的信任，并且被直接插入到你的网页中，因为一个怀有恶意的用户可能会使用这样的漏洞来做一些可能的坏事。这种类型的安全问题被叫做 跨站脚本（Cross Site Scripting） (XSS) 攻击。

为避免这个问题，你有两个选择：

- 第一， 你可以确保每一个不被信任的值都通过 `escape` 过滤器（下面的文档中将提到）运行，它将把潜在的有害HTML字符转换成无害的。This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
- 第二，你可以利用Django的自动HTML转义。 本节描述其余部分描述的是自动转义是如何工作的

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- `<` 会转换为 `<`
- `>` 会转换为 `>`
- `'` (单引号) 会转换为 `'`
- `"` (双引号)会转换为 `"`
- `&` 会转换为 `&`

我们要再次强调这是默认行为。如果你使用Django的模板系统，会处于保护之下。

如果关闭它

如果你不希望数据自动转义，在站点、模板或者变量级别，你可以使用几种方法来关闭它。

然而你为什么想要关闭它呢？由于有时，模板变量含有一些你打算渲染成原始HTML的数据，你并不想转义这些内容。例如，你可能会在数据库中储存一些HTML代码，并且直接在模板中嵌入它们。或者，你可能使用Django的模板系统来生成_不是_HTML的文本 -- 比如邮件信息。

用于独立变量

使用 `safe` 过滤器来关闭独立变量上的自动转移：

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

*safe*是*safe from further escaping*或者*can be safely interpreted as HTML*的缩写。在这个例子中，如果 `data` 含有 `'<b>'` ，输出会是：

```
This will be escaped: <b>
This will not be escaped: <b>
```

用于模板代码块

要控制模板上的自动转移，将模板（或者模板中的特定区域）包裹在 `autoescape` 标签 中，像这样：

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

`autoescape` 标签接受 `on` 或者 `off` 作为它的参数。有时你可能想在自动转移关闭的情况下强制使用它。下面是一个模板的示例：

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

自动转移标签作用于扩展了当前模板的模板，以及通过 `include` 标签包含的模板，就像所有block标签那样。例如：

base.html

```
{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
```

child.html

```
{% extends "base.html" %}
{% block title %}This &amp; that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

由于自动转义标签在base模板中关闭，它也会在child模板中关闭，导致当 `greeting` 变量含有 `<b>Hello!</b>` 字符串时，会渲染HTML。

```
<h1>This &amp; that</h1>
<b>Hello!</b>
```

注释

通常，模板的作用并不非常担心自动转义。Python一边的开发者（编写视图和自定义过滤器的人）需要考虑数据不应被转移的情况，以及合理地标记数据，让这些东西在模板中正常工作。

如果你创建了一个模板，它可能用于你不确定自动转移是否开启的环境，那么应该向任何需要转移的变量添加 `escape` 过滤器。当自动转移打开时， `escape` 过滤器双重过滤数据没有任何危险 -- `escape` 过滤器并不影响自动转义的变量。

字符串字面值和自动转义

像我们之前提到的那样，过滤器参数可以是字符串：

```
{{ data|default:"This is a string literal." }}
```

所有字面值字符串在插入模板时都 不会带有任何自动转义 -- 它们的行为类似于通过 `safe` 过滤器传递。背后的原因是，模板作者可以控制字符串字面值得内容，所以它们可以确保在模板编写时文本经过正确转义。

也即是说你可以编写

```
{{ data|default:"3 < 2" }}
```

...而不是：

```
{{ data|default:"3 < 2" }}  {# Bad! Don't do this. #}
```

这并不影响来源于模板自身的数据。模板内容在必要时仍然会自动转移，因为它们不受模板作者的控制。

访问方法调用

大多数对象上的方法调用同样可用于模板中。这意味着模板必须拥有对除了类属性（像是字段名称）和从视图中传入的变量之外的访问。例如，Django ORM提供了*“entry_set”*语法用于查找关联到外键的对象集合。所以，提供一个模型叫做“comment”，并带有一个关联到 “task” 模型的外键，你就可以遍历给定任务附带的所有评论，像这样：

```
{% for comment in task.comment_set.all %}
    {{ comment }}
{% endfor %}
```

与之类似，*QuerySets*提供了 `count()` 方法来计算含有对象的总数。因此，你可以像这样获取所有关于当前任务的评论总数：

```
{{ task.comment_set.all.count }}
```

当然，你可以轻易访问已经显式定义在你自己模型上的方法：

models.py

```
class Task(models.Model):
    def foo(self):
        return "bar"
```

template.html

```
{{ task.foo }}
```

由于Django有意限制了模板语言中逻辑处理的总数，不能够在模板中传递参数来调用方法。数据应该在视图中处理，然后传递给模板用于展示。

自定义标签和过滤器库

特定的应用提供自定义的标签和过滤器库。要在模板中访问它们，确保应用在 `INSTALLED_APPS` 之内（在这个例子中我们添加了 `'django.contrib.humanize'` ），之后在模板中使用 `load` 标签：

```
{% load humanize %}

{{ 45000|intcomma }}
```

上面的例子中， `load` 标签加载了 `humanize` 标签库，之后我们可以使用 `intcomma` 过滤器。如果你开启了 `django.contrib.admindocs` ，你可以查询admin站点中的文档部分，来寻找你的安装中的自定义库列表。

`load` 标签可以接受多个库名称，由空格分隔。例如：

```
{% load humanize i18n %}
```

关于编写你自己的自定义模板库，详见自定义模板标签和过滤器。

自定义库和模板继承

当你加载一个自定义标签或过滤器库时，标签或过滤器只在当前模板中有效 -- 并不是带有模板继承关系的任何父模板或者子模版中都有效。

例如，如果一个模板 `foo.html` 带有 `{% load humanize %}` ，子模版（例如，带有 `{% extends "foo.html" %}` ）中不能 访问humanize模板标签和过滤器。子模版需要添加自己的 `{% load humanize %}` 。

这个特性是可维护性和逻辑性的缘故。

另见

The Templates Reference

Covers built-in tags, built-in filters, using an alternative template, language, and more.

> 译者：Django 文档协作翻译小组，原文：Language overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

内置标签与过滤器

本文档描述的是django 内置模板标签和过滤器.我们推荐尽可能使用 自动文档，同时也可以自行编辑任何已安装的自定义标签或过滤器的文档。

内置标签参考指南

autoescape 自动转义

控制自动转义是否可用.这种标签带有任何 `on` 或 `off` 作为参数的话，他将决定转义块内效果。该标签会以一个 `endautoescape` 作为结束标签.

当自动转义生效时，所有变量内容会被转义成HTML输出（在所有过滤器生效后）这等同与手动将 `escape` 筛选器应用于每个变量。

唯一一个例外是，变量或者通过渲染变量的代码，或者因为它已经应用了 `safe` 或 `escape` 过滤器，已经被标记为“safe”。

样品用量：

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

块

block标签可以被子模板覆盖.查看 *Template inheritance* 可以获得更多信息.

评论

在 `{% comment %}` 和 `{% endcomment %}` ，之间的内容会被忽略，作为注释。在第一个标签可以插入一个可选的记录。 比如，当要注释掉一些代码时，可以用此来记录代码被注释掉的原因。

简单实例：

```
<p>Rendered text with {{ pub_date|date:"c" }}</p>
{% comment "Optional note" %}
    <p>Commented out text with {{ create_date|date:"c" }}</p>
{% endcomment %}
```

`comment` 标签不能嵌套使用。

csrf_token

这个标签用于跨站请求伪造保护，具体可以参考*Cross Site Request Forgeries*中的描述。

cycle

每当这个标签被访问,则传出一个它的可迭代参数的元素。第一次访问返回第一个元素,第二次访问返回第二个参数,以此类推.一旦所有的变量都被访问过了，就会回到最开始的地方，重复下去

这个标签在循环中特别有用:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

第一次迭代产生的HTML引用了 `row1` 类，第二次则是 `row2` 类，第三次 又是 `row1` 类，如此类推。

你也可以使用变量，例如，如果你有两个模版变量, `rowvalue1` 和 `rowvalue2` ,你可以让他们的值像这样替换:

```
{% for o in some_list %}
    <tr class="{% cycle rowvalue1 rowvalue2 %}">
        ...
    </tr>
{% endfor %}
```

被包含在cycle中的变量将会被转义。你可以禁止自动转义:

```
{% for o in some_list %}
    <tr class="{% autoescape off %}{% cycle rowvalue1 rowvalue2
%}{% endautoescape %}">
        ...
    </tr>
{% endfor %}
```

你能混合使用变量和字符串：

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' rowvalue2 'row3' %}">
        ...
    </tr>
{% endfor %}
```

In some cases you might want to refer to the current value of a cycle without advancing to the next value.To do this, just give the `{% cycle %}` tag a name, using “as”, like this:

```
{% cycle 'row1' 'row2' as rowcolors %}
```

From then on, you can insert the current value of the cycle wherever you’d like in your template by referencing the cycle name as a context variable.If you want to move the cycle to the next value independently of the original `cycle` tag, you can use another `cycle` tag and specify the name of the variable.所以，下面的模板：

```
<tr>
    <td class="{% cycle 'row1' 'row2' as rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
<tr>
    <td class="{% cycle rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
```

将输出：

```
<tr>
    <td class="row1">...</td>
    <td class="row1">...</td>
</tr>
<tr>
    <td class="row2">...</td>
    <td class="row2">...</td>
</tr>
```

`cycle` 标签中，通过空格分割，你可以使用任意数量的值。被包含在单引号 (`'`)或者双引号 (`"`) 中的值被认为是可迭代字符串，相反，没有被引号包围的值被当作模版变量。

默认情况下，当你在cycle标签中使用 `as` 关键字时，关于 `{% cycle %}` 的使用，会启动cycle并且直接产生第一个值。如果你想要在嵌套循环中或者included模版中使用这个值，那么将会遇到困难。如果你只是想要声明cycle，但是不产生第一

个值，你可以添加一个 `silent` 关键字来作为cycle标签的最后一个关键字。例如:

```
{% for obj in some_list %}
    {% cycle 'row1' 'row2' as rowcolors silent %}
    <tr class="{{ rowcolors }}">{% include "subtemplate.html" %}
</tr>
{% endfor %}
```

This will output a list of `<tr>` elements with `class` alternating between `row1` and `row2` .The subtemplate will have access to `rowcolors` in its context and the value will match the class of the `<tr>` that encloses it.If the `silent` keyword were to be omitted, `row1` and `row2` would be emitted as normal text, outside the `<tr>` element.

When the silent keyword is used on a cycle definition, the silence automatically applies to all subsequent uses of that specific cycle tag.The following template would output *nothing*, even though the second call to `{% cycle %}` doesn't specify `silent` :

```
{% cycle 'row1' 'row2' as rowcolors silent %}
{% cycle rowcolors %}
```

For backward compatibility, the `{% cycle %}` tag supports the much inferior old syntax from previous Django versions.You shouldn't use this in any new projects, but for the sake of the people who are still using it, here's what it looks like:

```
{% cycle row1,row2,row3 %}
```

In this syntax, each value gets interpreted as a literal string, and there's no way to specify variable values.Or literal commas.Or spaces.Did we mention you shouldn't use this syntax in any new projects?

debug

输出整个调试信息，包括当前上下文和导入的模块。

extends

表示当前模板继承自一个父模板

这个标签可以有两种用法:

- `{% extends "base.html" %}` (要有引号).继承名为 `"base.html"` 的父模板

- `{% extends variable %}` 使用 `variable` 的值. 如果变量被计算成一个字符串，Django将会把它看成是父模版的名字。如果变量被计算到一个 `Template` 对象，Django将会使用那个对象作为一个父模版。

阅读 *Template inheritance* 获取更多信息

filter

通过一个或多个过滤器对内容过滤。 作为灵活可变的语法，多个过滤器被管道符号相连接，且过滤器可以有参数。

注意块中所有的内容都应该包括在 `filter` 和 `endfilter` 标签中。

简单用例：

```
{% filter force_escape|lower %}
    This text will be HTML-escaped, and will appear in all lower
case.
{% endfilter %}
```

注意

`escape` 和 `safe` 过滤器不是可接受的参数。而应使用 `autoescape` 标记来管理模板代码块的自动转义。

firstof

输出第一个不为 `False` 参数。如果传入的所有变量都为 `False` ，就什么也不输出。

简单用例：

```
{% firstof var1 var2 var3 %}
```

它等价于：

```
{% if var1 %}
    {{ var1 }}
{% elif var2 %}
    {{ var2 }}
{% elif var3 %}
    {{ var3 }}
{% endif %}
```

当然你也可以用一个默认字符串作为输出以防传入的所有变量都是False：

```
{% firstof var1 var2 var3 "fallback value" %}
```

标签auto-escapes是开启的， 你可以这样关闭auto-escaping:

```
{% autoescape off %}
    {% firstof var1 var2 var3 "<strong>fallback value</strong>"
%}
{% endautoescape %}
```

如果只想要部分变量被规避，可以这样使用：

```
{% firstof var1 var2|safe var3 "<strong>fallback value</strong>"
|safe %}
```

for

循环组中的每一个项目，并让这些项目在上下文可用。 举个例子，展示 `athlete_list` 中的每个成员：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

可以利用 `{% for obj in list reversed %}` 反向完成循环。

如果你需要循环一个包含列表的列表，可以通过拆分每一个二级列表为一个独立变量来达到目的。 举个例子，如果你的内容包括一个叫做 `points` 的(x,y) 列表，你可以像以下例子一样输出points列表：

```
{% for x, y in points %}
    There is a point at {{ x }},{{ y }}
{% endfor %}
```

如果你想访问一个字典中的项目，这个方法同样有用。举个例子：如果你的内容包含一个叫做 `data` 的字典，下面的方式可以输出这个字典的键和值：

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

for循环设置了一系列在循环中可用的变量：

Variable	Description
`forloop.counter`	The current iteration of the loop (1-indexed)
`forloop.counter0`	The current iteration of the loop (0-indexed)
`forloop.revcounter`	The number of iterations from the end of the loop (1-indexed)
`forloop.revcounter0`	The number of iterations from the end of the loop (0-indexed)
`forloop.first`	True if this is the first time through the loop
`forloop.last`	True if this is the last time through the loop
`forloop.parentloop`	For nested loops, this is the loop surrounding the current one

for ... empty

`for` 标签带有一个可选的 `{% empty %}` 从句，以便在给出的组是空的或者没有被找到时，可以有所操作。

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

它和下面的例子作用相等，但是更简洁、更清晰甚至可能运行起来更快：

```
<ul>
  {% if athlete_list %}
    {% for athlete in athlete_list %}
      <li>{{ athlete.name }}</li>
    {% endfor %}
  {% else %}
    <li>Sorry, no athletes in this list.</li>
  {% endif %}
</ul>
```

if

`{% if %}` 会对一个变量求值，如果它的值是“True”（存在、不为空、且不是boolean类型的false值），这个内容块会输出：

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

上述例子中，如果 `athlete_list` 不为空，就会通过使用 `{{ athlete_list|length }}` 过滤器展示出运动员的数量。

正如你所见， `if` 标签之后可以带有一个或者多个 `{% elif %}` 从句，也可以带有一个 `{% else %}` 从句以便在之前的所有条件不成立的情况下完成执行。这些从句都是可选的。

布尔运算符

`if` 标签可以使用 `and` ， `or` 或 `not` 来测试多个变量或取消给定变量：

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches.
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

允许同时使用 `and` 和 `or` 子句， `and` 的优先级高于 `or` ：

```
{% if athlete_list and coach_list or cheerleader_list %}
```

将解释如下：

```
if (athlete_list and coach_list) or cheerleader_list
```

在 `if` 标记中使用实际括号是无效的语法。如果您需要它们指示优先级，则应使用嵌套的 `if` 标记。

`if` 标签还可能使用 `==`，`!=`，`<`，`>`，`<=`，`>=` 和 `in`，他们作用如下：

`==` 运算符

相等。例：

```
{% if somevar == "x" %}
  This appears if variable somevar equals the string "x"
{% endif %}
```

`!=` 运算符

不相等。例：

```
{% if somevar != "x" %}
  This appears if variable somevar does not equal the string "x",
  or if somevar is not found in the context
{% endif %}
```

`<` 运算符

小于。例：

```
{% if somevar < 100 %}
  This appears if variable somevar is less than 100.
{% endif %}
```

`>` 运算符

大于。例：

```
{% if somevar > 0 %}
  This appears if variable somevar is greater than 0.
{% endif %}
```

`<=` 运算符

小于或等于。例：

```
{% if somevar <= 100 %}
  This appears if variable somevar is less than 100 or equal to
100.
{% endif %}
```

`>=` 运算符

大于或等于。例：

```
{% if somevar >= 1 %}
  This appears if variable somevar is greater than 1 or equal to
 1.
{% endif %}
```

`in` 运算符

包含在内。许多Python容器支持此运算符，以测试给定值是否在容器中。以下是在 t> y中如何解释 `x 的一些示例：`

```
{% if "bc" in "abcdef" %}
  This appears since "bc" is a substring of "abcdef"
{% endif %}

{% if "hello" in greetings %}
  If greetings is a list or set, one element of which is the str
ing
  "hello", this will appear.
{% endif %}

{% if user in users %}
  If users is a QuerySet, this will appear if user is an
  instance that belongs to the QuerySet.
{% endif %}
```

`not in` 运算符

不包含在内。这是 `in` 运算符的否定操作。

比较运算符不能像Python或数学符号中那样“链接”。例如，不能使用：

```
{% if a > b > c %}  (WRONG)
```

你应该使用：

```
{% if a > b and b > c %}
```

过滤器

你也可以在 `if` 表达式中使用过滤器。举个例子：

```
{% if messages|length >= 100 %}
   You have lots of messages today!
{% endif %}
```

复合表达式

所有上述操作符可以组合以形成复杂表达式。对于这样的表达式，重要的是要知道在表达式求值时如何对运算符进行分组 - 即优先级规则。操作符的优先级从低至高如下：

- or
- and
- not
- in 中
- ==，!=，<，>，<=，>=

（这完全依据Python）。所以，例如，下面的复杂 `if` 标签：

```
{% if a == b or c == d and e %}
```

...将被解释为：

```
(a == b) or ((c == d) and e)
```

如果你想要不同的优先级，那么你需要使用嵌套的 `if` 标签。有时，为了清楚起见，更好的是为了那些不知道优先规则的人。

ifchanged

检查一个值是否在上一次的迭代中改变。

`{% ifchanged %}` 块标签用在循环里。它可能有两个用处：

1. 检查它已经渲染过的内容中的先前状态。并且只会显示发生改变的内容。例如，以下的代码是输出days的列表项，不过它只会输出被修改过月份的项:

```
&lt;h1&gt;Archive for {{ year }}&lt;/h1&gt;

{% for date in days %}
    {% ifchanged %}&lt;h3&gt;{{ date|date:"F" }}&lt;/h3&gt;{
% endifchanged %}
    &lt;a href="{{ date|date:"M/d"|lower }}/"&gt;{{ date|dat
e:"j" }}&lt;/a&gt;
{% endfor %}
```

2. 如果标签内被给予多个值时,则会比较每一个值是否与上一次不同。例如，以下显示每次更改时的日期，如果小时或日期已更改，则显示小时：

```
{% for date in days %}
    {% ifchanged date.date %} {{ date.date }} {% endifchange
d %}
    {% ifchanged date.hour date.date %}
        {{ date.hour }}
    {% endifchanged %}
{% endfor %}
```

`ifchanged` 标记也可以采用可选的 `{% else %}` 将显示如果值没有改变：

```
{% for match in matches %}
    <div style="background-color:
  {% ifchanged match.ballot_id %}
  {% cycle "red" "blue" %}
  {% else %}
 gray
  {% endifchanged %}
 ">{{ match }}</div>
{% endfor %}
```

ifequal

如果给定的两个参数是相等的,则显示被标签包含的内容.

举个例子:

```
{% ifequal user.pk comment.user_id %}
    ...
{% endifequal %}
```

在 `if` 标签里面, 也可以包含 `{% else %}` 标签选项.

参数可以是一个硬编码的字符串,所以也可以这样:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

`ifequal` 标记的替代方法是使用 `if` 标记和 `==` 运算符。

ifnotequal

就像 `ifequal` ，不过它测试两个参数不相等。

`ifnotequal` 标记的替代方法是使用 `if` 标记和 `!=` 运算符。

include

加载模板并以标签内的参数渲染。这是一种可以引入别的模板的方法。

模板名可以是变量或者是硬编码的字符串，可以用单引号也可以是双引号.

下面这个示例包括模板 `“foo/bar.html”` 的内容：

```
{% include "foo/bar.html" %}
```

此示例包括其名称包含在变量 `template_name` 中的模板的内容：

```
{% include template_name %}
```

Changed in Django 1.7:

变量也可以是任何实现了 `render()` 方法接口的对象，这个对象要可以接收上下文（context）。这就允许你在context中引用一个已经被编译过的 `Template` 。

被包含的模板在包含它的模板的上下文中渲染。下面这个示例生成输出 `“Hello, John!”` ：

- 上下文：变量 `person` 设置为 `“John”` ，变量 `greeting` 设置为 `“Hello”` 。

- 模板：

  ```
  {% include "name_snippet.html" %}
  ```

- `name_snippet.html` 模板：

  ```
  {{ greeting }}, {{ person|default:"friend" }}!
  ```

你可以使用关键字参数将额外的上下文传递到模板：

```
{% include "name_snippet.html" with person="Jane" greeting="Hello" %}
```

如果要仅使用提供的变量（或根本不使用变量）来渲染上下文，请使用 `only` 选项。所包含的模板没有其他变量可用：

```
{% include "name_snippet.html" with greeting="Hi" only %}
```

注意

`include` 标签应该被理解为是一种"将子模版渲染并嵌入HTML中"的变种方法,而不是认为是"解析子模版并在被父模版包含的情况下展现其被父模版定义的内容".这意味着在不同的被包含的子模版之间并不共享父模版的状态,每一个子包含都是完全独立的渲染过程.

Block模块在被包含 之前 就已经被执行. 这意味着模版在被包含之前就已经从另一个block扩展并 已经被执行并完成渲染 - 没有block模块会被include引入并执行,即使父模版中的扩展模版.

load

加载自定义模板标签集。

举个例子, 下面这模板将会从 `package` 包中载入所有 `somelibrary` 和 `otherlibrary` 中已经注册的标签和过滤器:

```
{% load somelibrary package.otherlibrary %}
```

你还可以使用 `from` 参数从库中选择性加载单个过滤器或标记。在下面这个示例中，名为 `foo` 和 `bar` 的模板标签/过滤器将从 `somelibrary` 加载：

```
{% load foo bar from somelibrary %}
```

有关详细信息，请参阅自定义标记和过滤器库。

lorem

New in Django 1.8:

该标签之前位于 `django.contrib.webdesign` 中。

展示随机的“lorem ipsum”拉丁文本. 这个标签是用来在模版中提供文字样本以供测试用的.

用法：

```
{% lorem [count] [method] [random] %}
```

可以使用零个，一个，两个或三个参数使用 `{% lorem %}` 。这些参数是：

Argument	Description
`count`	A number (or variable) containing the number of paragraphs or words to generate (default is 1).
`method`	Either `w` for words, `p` for HTML paragraphs or `b` for plain-text paragraph blocks (default is `b`).
`random`	The word `random` , which if given, does not use the common paragraph (“Lorem ipsum dolor sit amet...”) when generating text.

例子：

- `{% lorem %}` 将输出常见的“lorem ipsum”段落。
- `{% lorem 3 p %} 输出常见的“lorem ipsum”段落和两个随机段落，每个段落包` <p> 标签中。
- `{% lorem 2 w 随机 %} / t6>` 将输出两个随机拉丁字。

now

显示最近的日期或事件,可以通过给定的字符串格式显示。此类字符串可以包含格式说明符字符，如 `date` 过滤器部分中所述。

例：

```
It is {% now "jS F Y H:i" %}
```

注意！，如果你想要使用“raw”值，你能够反斜杠转义一个格式化字符串。在这个例子中，“o”和“f”都是反斜杠转义，因为如果不这样，会分别显示年和时间：

```
It is the {% now "jS \o\f F" %}
```

这将显示为“这是9月4日”。

注意

传递的格式也可以是预定义的 `DATE_FORMAT` ， `DATETIME_FORMAT` ， `SHORT_DATE_FORMAT` 或 `SHORT_DATET` 之一。预定义的格式可能会因当前语言环境和格式本地化的启用而有所不同，例如：

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

You can also use the syntax `{% now "Y" as current_year %}` to store the output inside a variable.This is useful if you want to use `{% now %}` inside a template tag like `blocktrans` for example:

```
{% now "Y" as current_year %}
{% blocktrans %}Copyright {{ current_year }}{% endblocktrans %}
```

New in Django 1.8.

添加了使用“as”语法的能力。

regroup

用相似对象间共有的属性重组列表.

用一个例子来解释这种复杂的标签是最好的方法:say that “places” is a list of cities represented by dictionaries containing `"name"` , `"population"` , and `"country"` keys:

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'I
ndia'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country':
'India'},
    {'name': 'New York', 'population': '20,000,000', 'country':
'USA'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'U
SA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Ja
pan'},
]
```

你会希望用下面这种方式来展示国家和城市的信息

- India
  - 孟买：19,000,000
  - 加尔各答：15,000,000
- USA
  - 纽约：20,000,000
  - 芝加哥：7,000,000
- Japan
  - 东京：33,000,000

你可以使用 `{% regroup %}` 标签来给每个国家的城市分组。以下模板代码片段将实现这一点：

```
{% regroup cities by country as country_list %}

<ul>
{% for country in country_list %}
    <li>{{ country.grouper }}
    <ul>
        {% for item in country.list %}
          <li>{{ item.name }}: {{ item.population }}</li>
        {% endfor %}
    </ul>
    </li>
{% endfor %}
</ul>
```

让我们来看看这个例子。 `{% regroup %}` 有三个参数：你想要重组的列表，被分组的属性，还有结果列表的名字. 在这里，我们通过 `country` 属性重新分组 `cities` 列表，并调用结果 `country_list` 。

`{% regroup %}` 产生一个清单（在本例中为 `country_list` 的组对象。每个组对象有两个属性：

- `grouper` - 按分组的项目（例如，字符串“India”或“Japan”）。
- `list` - 此群组中所有项目的列表（例如，所有城市的列表，其中country ='India'）。

请注意， `{% regroup %}` 不会对其输入进行排序！我们的例子依赖于事实： `cities` 列表首先由 `country` 排序。如果 `cities` 列表不通过 `country` 对其成员进行排序，则重新分组将天真显示单个国家/地区的多个组。例如，假设 `cities` 列表已设置为此（请注意，国家/地区未分组在一起）：

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

对于 `cities` 的输入，示例 `{% regroup %}` 以上将导致以下输出：

- India
    - 孟买：19,000,000
- USA
    - 纽约：20,000,000
- India
    - 加尔各答：15,000,000
- USA
    - 芝加哥：7,000,000
- Japan
    - 东京：33,000,000

这个问题的最简单的解决方案是确保在你的视图代码中，数据是根据你想要显示的顺序排序。

另一个解决方案是使用 `dictsort` 过滤器对模板中的数据进行排序，如果您的数据在字典列表中：

```
{% regroup cities|dictsort:"country" by country as country_list %}
```

Grouping on other properties

一个有效的模版查找是一个regroup标签的合法的分组属性。包括方法，属性，字典健和列表项。例如，如果“country”字段是具有属性“description”的类的外键，则可以使用：

```
{% regroup cities by country.description as country_list %}
```

或者，如果 `country` 是具有 `choices` 的字段，则它将具有作为属性的 `get_FOO_display()` 方法，显示字符串而不是 `choices` 键：

```
{% regroup cities by get_country_display as country_list %}
```

`{{ country.grouper }}` 现在会显示 `choices`

spaceless

删除HTML标签之间的空白格.包括制表符和换行.

用法示例：

```
{% spaceless %}
    <p>
        <a href="foo/">Foo</a>
    </p>
{% endspaceless %}
```

这个示例将返回下面的HTML：

```
<p><a href="foo/">Foo</a></p>
```

仅删除 *tags* 之间的空格 – 而不是标签和文本之间的。在此示例中， `Hello` 周围的空格不会被删除：

```
{% spaceless %}
    <strong>
        Hello
    </strong>
{% endspaceless %}
```

ssi

从1.8以后不建议使用这标签建议不再使用，将会在Django 2.0以后会被删除。用 `include` 标签代替。

将给定文件的内容输出到页面。

像一个简单的 `include` 标签， `{% ssi %}` 在当前页面中的另一个文件 - 必须使用绝对路径指定：

```
{% ssi '/home/html/ljworld.com/includes/right_generic.html' %}
```

`ssi` 的第一个参数可以是引用的文字或任何其他上下文变量。

如果给出了可选的 `parsed` 参数，则在当前上下文中，将包含文件的内容作为模板代码进行评估：

```
{% ssi '/home/html/ljworld.com/includes/right_generic.html' pars
ed %}
```

Note that if you use `{% ssi %}`, you'll need to define `'allowed_include_roots'` in the `OPTIONS` of your template engine, as a security measure.

注意

使用 `ssi` 标记和 `parsed` 参数，文件之间没有共享状态 - 每个包含是一个完全独立的呈现过程。这意味着例如不可能使用包含的文件来定义块或改变当前页面中的上下文。

另请参阅：`{% include %}`。

templatetag

输出用于构成模板标记的语法字符之一。

由于模板系统没有“转义”的概念，为了显示模板标签中使用的一个位，必须使用 `{% templatetag %}` 标记。

参数指定要输出哪个模板位：

Argument	Outputs
`openblock`	`{%`
`closeblock`	`%}`
`openvariable`	`{{`
`closevariable`	`}}`
`openbrace`	`{`
`closebrace`	`}`
`opencomment`	`{#`
`closecomment`	`#}`

样品用量：

```
{% templatetag openblock %} url 'entry_list' {% templatetag clos
eblock %}
```

url

返回一个绝对路径的引用(不包含域名的URL)，该引用匹配一个给定的视图函数和一些可选的参数。在解析后返回的结果路径字符串中，每个特殊字符将使用 `iri_to_uri()` 编码。

这是一种不违反DRY原则的输出链接的方式，它可以避免在模板中硬编码链接路径。

```
{% url 'some-url-name' v1 v2 %}
```

第一个参数是视图函数中包名.模块名.函数名这样的路径 `package.package.module.function` . 它可以是一个被引号引起来的字符串或者其他的上下文变量. 其他参数是可选的并且应该以空格隔开，这些值会在URL中以参数的形式传递. 上面的例子展示了如何传递位置参数.当然你也可以使用关键字参数.

```
{% url 'some-url-name' arg1=v1 arg2=v2 %}
```

不要把位置参数和关键字参数混在一起使用。URLconf所需的所有参数都应该存在。

例如，假设您有一个视图 `app_views.client` ，其URLconf接受客户端ID（此处 `client()` 是视图文件 `app_views .py` ）。URLconf行可能如下所示：

```
('^client/([0-9]+)/$', 'app_views.client', name='app-views-clien
t')
```

如果你的应用中的URLconf 已经被包含到项目 URLconf 中，比如下面这样

```
('^clients/', include('project_name.app_name.urls'))
```

...那么, 在模板文件中, 你可以很方便的创建一个接指该视图的超链接，示例如下

```
{% url 'app-views-client' client.id %}
```

模板标签会输出如下的字符串 `/clients/client/123/` .

如果您使用命名的网址格式，则可以参考 `网址` 标记中的模式名称，而不是使用视图的路径。

请注意，如果您要撤消的网址不存在，您会收到 `NoReverseMatch` 异常，这会导致您的网站显示错误网页。

如果您希望在不显示网址的情况下检索网址，则可以使用略有不同的调用：

```
{% url 'some-url-name' arg arg2 as the_url %}

<a href="{{ the_url }}">I'm linking to {{ the_url }}</a>
```

The scope of the variable created by the `as var` syntax is the `{% block %}` in which the `{% url %}` tag appears.

此 `{% url ...` 为var%}语法将不导致错误，如果视图丢失。实际上，您将使用此链接来链接到可选的视图：

```
{% url 'some-url-name' as the_url %}
{% if the_url %}
  <a href="{{ the_url }}">Link to optional stuff</a>
{% endif %}
```

如果您要检索名称空间网址，请指定完全限定名称：

```
{% url 'myapp:view-name' %}
```

这将遵循正常的命名空间URL解析策略，包括使用上下文对当前应用程序提供的任何提示。

自1.8版起已弃用：点状Python路径语法已弃用，将在Django 2.0中删除：

```
{% url 'path.to.some_view' v1 v2 %}
```

警告

不要忘记在函数路径或模式名称周围加引号，否则值将被解释为上下文变量！

verbatim

停止模版引擎在该标签中的渲染/

常见的用法是允许与Django语法冲突的JavaScript模板图层。例如：

```
{% verbatim %}
    {{if dying}}Still alive.{{/if}}
{% endverbatim %}
```

You can also designate a specific closing tag, allowing the use of `{% endverbatim %}` as part of the unrendered contents:

```
{% verbatim myblock %}
    Avoid template rendering via the {% verbatim %}{% endverbati
m %} block.
{% endverbatim myblock %}
```

widthratio

为了创建条形图等，此标签计算给定值与最大值的比率，然后将该比率应用于常量。

例如：

```
<img src="bar.png" alt="Bar"
     height="10" width="{% widthratio this_value max_value max_w
idth %}" />
```

如果 this_value 是175， max_value 是200，并且 max_width 是100，则上述示例中的图像将是88像素宽（因为175 / 200 = .875； .875 * 100 = 87.5，上舍入为88）。

在某些情况下，您可能想要捕获变量中的 widthratio 的结果。它可以是有用的，例如，在 blocktrans 像这样：

```
{% widthratio this_value max_value max_width as width %}
{% blocktrans %}The width is: {{ width }}{% endblocktrans %}
```

Changed in Django 1.7:

添加了使用“as”与此标记一样的能力，如上例所示。

与

使用一个简单地名字缓存一个复杂的变量，当你需要使用一个“昂贵的”方法（比如访问数据库）很多次的时候是非常有用的

例如：

```
{% with total=business.employees.count %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

{% 与 %}之间可用填充变量（上例中 total t5> 和 {% endwith %}

你可以分配多个上下文变量：

```
{% with alpha=1 beta=2 %}
    ...
{% endwith %}
```

注意

The previous more verbose format is still supported: `{% with business.employees.count as total %}`

内置过滤器参考

加

把add后的参数加给value

例如:

```
{{ value|add:"2" }}
```

如果 `value` 为 `4`,则会输出 `6`.

过滤器首先会强制把两个值转换成Int类型。如果强制转换失败, 它会试图使用各种方式吧两个值相加。它会使用一些数据类型 (字符串, 列表, 等等.) 其他类型则会失败. 如果转换失败，结果会变成一个空字符串

例如，我们使用下面的值

```
{{ first|add:second }}
```

`first` 是 `[1, 2, 3]` ，`second` 是 `[4, 5, 6]`, 将会输出 `[1, 2, 3, 4, 5, 6]`.

警告

如果字符串可以被强制转换成int类型则会 **summed**，无法被转换，则和上面的第一个例子一样

addslashes

在引号前面加上斜杆。例如，用于在CSV中转义字符串。

例如：

```
{{ value|addslashes }}
```

如果 `value` 是 `"I'm using Django"`, 输出将变成 `"I\'m using Django"`.

capfirst

大写变量的第一个字母。如果第一个字符不是字母，该过滤器将不会生效。

例如：

```
{{ value|capfirst }}
```

如果 `value` 是 `"django"`, 输出将变成 `"Django"`.

中央

使"value"在给定的宽度范围内居中.

例如:

```
"{{ value|center:"15" }}"
```

如果 `value` 是 `"Django"`，输出将是 `“ Django t7>` 。

切

移除value中所有的与给出的变量相同的字符串

例如：

```
{{ value|cut:" " }}
```

如果 `value` 为 `“String 与 空格”`，输出将为 `"Stringwithspaces"` 。

日期

根据给定格式对一个date变量格式化

格式类似于 PHP 的 `date()` 函数 (http://php.net/date) ，在一些细节上有不同.

注意

这些格式字符不在模板外的Django中使用。它们被设计为与PHP兼容，以便为设计者轻松过渡。

可用的格式字符串：

Format character	Description	Example output
a	`'a.m.'` or `'p.m.'` (Note that this is slightly different than PHP's output, because this includes periods to match Associated Press style.)	`'a.m.'`
A	`'AM'` or `'PM'` .	`'AM'`
b	Month, textual, 3 letters, lowercase.	`'jan'`
B	Not implemented.	
c	ISO 8601 format. (Note: unlike others formatters, such as "Z", "O" or "r", the "c" formatter will not add timezone offset if value is a naive datetime (see `datetime.tzinfo`).	`2008-01-02T10:30:00.000123+02:00` , or `2008-01-02T10:30:00.000123` if the datetime is naive
d	Day of the month, 2 digits with leading zeros.	`'01'` to `'31'`
D	Day of the week, textual, 3 letters.	`'Fri'`
e	Timezone name. Could be in any format, or might return an empty string, depending on the datetime.	`''` , `'GMT'` , `'-500'` , `'US/Eastern'` , etc.
E	Month, locale specific alternative representation usually used for long date representation.	`'listopada'` (for Polish locale, as opposed to `'Listopad'`)
	Time, in 12-hour hours and minutes,	

f	with minutes left off if they're zero. Proprietary extension.	'1' , '1:30'
F	Month, textual, long.	'January'
g	Hour, 12-hour format without leading zeros.	'1' to '12'
G	Hour, 24-hour format without leading zeros.	'0' to '23'
h	Hour, 12-hour format.	'01' to '12'
H	Hour, 24-hour format.	'00' to '23'
i	Minutes.	'00' to '59'
I	Daylight Savings Time, whether it's in effect or not.	'1' or '0'
j	Day of the month without leading zeros.	'1' to '31'
l	Day of the week, textual, long.	'Friday'
L	Boolean for whether it's a leap year.	True or False
m	Month, 2 digits with leading zeros.	'01' to '12'
M	Month, textual, 3 letters.	'Jan'
n	Month without leading zeros.	'1' to '12'
N	Month abbreviation in Associated Press style. Proprietary extension.	'Jan.' , 'Feb.' , 'March' , 'May'
o	ISO-8601 week-numbering year, corresponding to the ISO-8601 week number (W)	'1999'
O	Difference to Greenwich time in hours.	'+0200'

P	Time, in 12-hour hours, minutes and 'a.m.'/'p.m.', with minutes left off if they're zero and the special-case strings 'midnight' and 'noon' if appropriate. Proprietary extension.	`'1 a.m.'` , `'1:30 p.m.'` , `'midnight'` , `'noon'` , `'12:30 p.m.'`
r	RFC 2822 formatted date.	`'Thu, 21 Dec 2000 16:01:07 +0200'`
s	Seconds, 2 digits with leading zeros.	`'00'` to `'59'`
S	English ordinal suffix for day of the month, 2 characters.	`'st'` , `'nd'` , `'rd'` or `'th'`
t	Number of days in the given month.	`28` to `31`
T	Time zone of this machine.	`'EST'` , `'MDT'`
u	Microseconds.	`000000` to `999999`
U	Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC).	
w	Day of the week, digits without leading zeros.	`'0'` (Sunday) to `'6'` (Saturday)
W	ISO-8601 week number of year, with weeks starting on Monday.	`1` , `53`
y	Year, 2 digits.	`'99'`
Y	Year, 4 digits.	`'1999'`
z	Day of the year.	`0` to `365`
Z	Time zone offset in seconds. The offset for timezones west of UTC is always	`-43200` to `43200`

	negative, and for those east of UTC is always positive.	

例如：

```
{{ value|date:"D d M Y" }}
```

如果 `value` 是 `datetime` 对象（例如，`datetime.datetime.now()` 的结果），输出将是字符串 `'Wed 09 Jan 2008'`。

传递的格式可以是预定义的格式 `DATE_FORMAT`，`DATETIME_FORMAT`，`SHORT_DATE_FORMAT` 或 `SHORT_DATET` 使用上表中显示的格式说明符。请注意，预定义的格式可能会根据当前语言环境而有所不同。

假设 `USE_L10N` 为 `True` 和 `LANGUAGE_CODE` 为例如 `"es"`

```
{{ value|date:"SHORT_DATE_FORMAT" }}
```

the output would be the string `"09/01/2008"` (the `"SHORT_DATE_FORMAT"` format specifier for the `es` locale as shipped with Django is `"d/m/Y"`).

不使用格式字符串时使用：

```
{{ value|date }}
```

...将使用 `DATE_FORMAT` 设置中定义的格式化字符串，而不应用任何本地化。

您可以将 `date` 与 `time` 过滤器结合使用，以呈现 `datetime` 值的完整表示形式。例如。：

```
{{ value|date:"D d M Y" }} {{ value|time:"H:i" }}
```

默认

如果value的计算结果为 `False`，则使用给定的默认值。否则，使用该value。

例如：

```
{{ value|default:"nothing" }}
```

如果 `value` 为 `""`（空字符串），则输出将为 `nothing`。

default_if_none

如果（且仅当）value为 `None` ，则使用给定的默认值。否则，使用该value。

注意，如果给出一个空字符串，默认值将不被使用。如果要回退空字符串，请使用 `default` 过滤器。

例如：

```
{{ value|default_if_none:"nothing" }}
```

如果 `value` 为 `None` ，则输出将为字符串 `“nothing”` 。

dictsort

接受一个字典列表，并返回按参数中给出的键排序后的列表。

例如：

```
{{ value|dictsort:"name" }}
```

如果 `value` 为：

```
[
    {'name': 'zed', 'age': 19},
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
]
```

那么输出将是：

```
[
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
    {'name': 'zed', 'age': 19},
]
```

你也可以做更复杂的事情，如：

```
{% for book in books|dictsort:"author.age" %}
    * {{ book.title }} ({{ book.author.name }})
{% endfor %}
```

如果 `books` 是：

```
[
    {'title': '1984', 'author': {'name': 'George', 'age': 45}},
    {'title': 'Timequake', 'author': {'name': 'Kurt', 'age': 75}
},
    {'title': 'Alice', 'author': {'name': 'Lewis', 'age': 33}},
]
```

那么输出将是：

```
* Alice (Lewis)
* 1984 (George)
* Timequake (Kurt)
```

dictsort翻转

获取字典列表，并返回按照参数中给出的键按相反顺序排序的列表。这与上面的过滤器完全相同，但返回的值将是相反的顺序。

可分割

如果value可以被给出的参数整除，则返回 `True`

例如：

```
{{ value|divisibleby:"3" }}
```

如果 `value` 是 `21` ，则输出将为 `True` 。

逃逸

转义字符串的HTML。具体来说，它使这些替换：

- `<` 转换为 `<`
- `>` 转换为 `>`
- `'` （单引号）转换为 `'`
- `"` （双引号）转换为 `"`
- `&` 转换为 `&`

转义仅在字符串输出时应用，因此在连接的过滤器序列中 `escape` 的位置无关紧要：它将始终应用，就像它是最后一个过滤器。如果要立即应用转义，请使用 `force_escape` 过滤器。

将 `转义` 应用于通常会对结果应用自动转义的变量只会导致一轮转义完成。因此，即使在自动逃逸环境中使用此功能也是安全的。如果要应用多个转义通过，请使用 `force_escape` 过滤器。

例如，您可以在 `autoescape` 关闭时将 `escape` 应用于字段：

```
{% autoescape off %}
    {{ title|escape }}
{% endautoescape %}
```

escapejs

转义用于JavaScript字符串的字符。这使不使字符串安全用于HTML，但确保在使用模板生成JavaScript / JSON时避免语法错误。

例如：

```
{{ value|escapejs }}
```

如果 `value` 为 “testing \ r \ njavascript \'string” &lt； b&gt； escapi，输出将为 “testing \\ u000D \\ u000Ajavascript \\ u0027string \\ u0022 \\ u0。

filesizeformat

格式化数值为“人类可读”的文件大小（例如 `'13 KB'`, `'4.1 MB'`, `'102 bytes'` 等）。

例如：

```
{{ value|filesizeformat }}
```

如果 `value` 为123456789，输出将是 `117.7 MB` 。

文件大小和国际系统单位

严格地讲， `filesizeformat` 没有遵守国际单位系统建议的KiB、MiB、GiB等，它们使用1024 为幂（虽然这里使用的也是）。相反，Django 使用传统的更常用的单位名称（KB、MB、GB等）。

第一

返回列表中的第一项。

例如：

```
{{ value|first }}
```

如果 `值` 是列表 `['a'， 'b'， 'c']` ，输出将为 `'a'` 。

floatformat

当不使用参数时，将浮点数舍入到小数点后一位，但前提是要显示小数部分。例如：

`value`	Template	Output
`34.23234`	`{{ value|floatformat }}`	`34.2`
`34.00000`	`{{ value|floatformat }}`	`34`
`34.26000`	`{{ value|floatformat }}`	`34.3`

如果与数字整数参数一起使用， `floatformat` 将数字四舍五入为小数位数。例如：

`value`	Template	Output
`34.23234`	`{{ value|floatformat:3 }}`	`34.232`
`34.00000`	`{{ value|floatformat:3 }}`	`34.000`
`34.26000`	`{{ value|floatformat:3 }}`	`34.260`

特别有用的是传递0（零）作为参数，它将使float浮动到最接近的整数。

`value`	Template	Output
`34.23234`	`{{ value|floatformat:"0" }}`	`34`
`34.00000`	`{{ value|floatformat:"0" }}`	`34`
`39.56000`	`{{ value|floatformat:"0" }}`	`40`

如果传递给 `floatformat` 的参数为负，则它会将一个数字四舍五入到小数点后的位置，但前提是要显示一个小数部分。例如：

`value`	Template	Output
`34.23234`	`{{ value|floatformat:"-3" }}`	`34.232`
`34.00000`	`{{ value|floatformat:"-3" }}`	`34`
`34.26000`	`{{ value|floatformat:"-3" }}`	`34.260`

使用没有参数的 `floatformat` 等效于使用具有 `-1` 的参数的 `floatformat` 。

force_escape

将HTML转义应用于字符串（有关详细信息，请参阅 `escape` 过滤器）。此过滤器立即应用于，并返回一个新的转义字符串。这在需要多次转义或想要对转义结果应用其他过滤器的罕见情况下非常有用。通常，您要使用 `escape` 过滤器。

例如，如果您要捕获由 `linebreaks` 过滤器创建的 `<p>` HTML元素：

```
{% autoescape off %}
    {{ body|linebreaks|force_escape }}
{% endautoescape %}
```

get_digit

给定一个整数，返回所请求的数字，其中1是最右边的数字，2是第二个最右边的数字等。返回无效输入的原始值（如果输入或参数不是整数，或参数小于1）。否则，输出总是一个整数。

例如：

```
{{ value|get_digit:"2" }}
```

如果 `value` 为 `123456789` ，则输出将为 `8` 。

iriencode

将IRI（国际化资源标识符）转换为适合包含在URL中的字符串。如果您尝试在网址中使用包含非ASCII字符的字符串，这是必要的。

在已经通过 `urlencode` 过滤器的字符串上使用此过滤器是安全的。

例如：

```
{{ value|iriencode }}
```

如果 `value` 为 `"?test=1&me=2"` ，输出将为 `"?test=1&me=2"`

加入

使用字符串连接列表，例如Python的 `str.join(list)`

例如：

```
{{ value|join:" // " }}
```

如果 `value` 是列表 `['a'， 'b'， 'c'] / t2>，输出将是字符串` “a // b // c“ `。`

持续

返回列表中的最后一个项目。

例如：

```
{{ value|last }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output will be the string `"d"`.

长度

返回值的长度。这适用于字符串和列表。

例如：

```
{{ value|length }}
```

如果 `value` 是 `['a'， 'b'， 'c'， 'd']` 或 `"abcd"` ，输出将为 `4` 。

Changed in Django 1.8:

对于未定义的变量，过滤器返回 `0` 。以前，它返回一个空字符串。

length_is

如果值的长度是参数，则返回 `True` ，否则返回 `False` 。

例如：

```
{{ value|length_is:"4" }}
```

如果 `value` 是 `['a'， 'b'， 'c'， 'd']` 或 `"abcd"` ，输出将为 `True` 。

linebreaks

用适当的HTML替换纯文本中的换行符；单个换行符变为HTML换行符（ `&lt； br /&gt；` ），新行后跟空行将成为段落（ `</p>` ）。

例如：

```
{{ value|linebreaks }}
```

如果`value`为`Joel为 a slug`，输出将为`&lt； p&gt； Joel&lt； br /&gt；是 a slug&lt； / p&gt；`

linebreaksbr

将纯文字中的所有换行符转换为HTML换行符（`&lt； br /&gt；`）。

例如：

```
{{ value|linebreaksbr }}
```

如果`value`为`Joel为 a slug`，输出将为`Joel&lt； br /&gt；是 a slug`

亚麻布

显示带行号的文本。

例如：

```
{{ value|linenumbers }}
```

如果`value`为：

```
one
two
three
```

输出将是：

```
1\. one
2\. two
3\. three
```

ljust

将给定宽度的字段中的值左对齐。

参数：字段大小

例如：

```
"{{ value|ljust:"10" }}"
```

如果 `value` 为 `Django` ，则输出将为 `“Django ”` 。

降低

将字符串转换为全部小写。

例如：

```
{{ value|lower }}
```

如果 `value` 为 `仍然 MAD 在 Yoko 输出将在` 仍然 mad 在 yoko ` 。`

make_list

返回转换为列表的值。对于字符串，它是一个字符列表。对于整数，在创建列表之前将参数强制转换为unicode字符串。

例如：

```
{{ value|make_list }}
```

如果 `value` 是字符串 `"Joel"` ，输出将是列表 `['J'， 'o' ， 'e'， 'l']` 。如果 `value` 是 `123` ，则输出将是列表 `['1'， '2'， t15 > '3']` 。

phone2numeric

将电话号码（可能包含字母）转换为其等效数字。

输入不必是有效的电话号码。这将很乐意转换任何字符串。

例如：

```
{{ value|phone2numeric }}
```

如果 `value` 为 `800-COLLECT` ，输出将为 `800-2655328` 。

复数

如果值不是1则返回一个复数形式通常用 `'s'` 表示.

例：

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

如果`num_messages`的值是`1`，那么将会输出`You have 1 message.` 如果`num_messages`的值是`2`那么将会输出`You have 2 messages.`

另外如果你需要的不是`'s'`后缀的话，你可以提供一个备选的参数给过滤器

例：

```
You have {{ num_walruses }} walrus{{ num_walruses|pluralize:"es"
}}.
```

对于非一般形式的复数,你可以同时指定 单复数形式，用逗号隔开.

例：

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ie
s" }}.
```

注意

使用`blocktrans`来翻译复数形式的字符串

打印

包装器`pprint.pprint()` - 用于调试，真的。

随机

返回给定列表中的随机项。

例如：

```
{{ value|random }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output could be `"b"`.

removetags

自1.8版起已弃用：`removetags`无法保证HTML安全输出，因安全问题而被弃用。请考虑使用漂白。

从输出中删除[X] HTML标签的空格分隔列表。

例如：

```
{{ value|removetags:"b span" }}
```

If `value` is `"<b>Joel</b> <button>is</button> a <span`
the unescaped output will be
`"Joel <button>is</button> a slug"` .

请注意，此过滤器区分大小写。

If `value` is `"<B>Joel</B> <button>is</button> a <span`
the unescaped output will be
`"<B>Joel</B> <button>is</button> a slug"` .

无安全保证

请注意， `removetags` 不会保证其输出是HTML安全的。特别地，它不递归地工作，因此像 `"<sc<script>ript>alert('XSS')</sc</script>rip` 即使您应用 `|removetags:"script"` 也是安全的。因此，如果输入是用户提供的，则**NEVER**将 `safe` 过滤器应用于 `removetags` 输出。如果您正在寻找更强大的功能，可以使用 `bleach` Python库，特别是其清洁方法。

rjust

右对齐给定宽度字段中的值。

参数：字段大小

例如：

```
"{{ value|rjust:"10" }}"
```

如果 `value` 为 `Django` ，则输出将为 `“ Django”` 。

安全

将字符串标记为在输出之前不需要进一步的HTML转义。当自动转义关闭时，此过滤器不起作用。

注意

如果您要链接过滤器，在 `safe` 后应用的过滤器可能会使内容再次不安全。例如，以下代码按原样打印变量：

```
{{ var|safe|escape }}
```

safeseq

将 `safe` 过滤器应用于序列的每个元素。与对序列进行操作的其他过滤器（例如 `join` ）一起使用非常有用。例如：

```
{{ some_list|safeseq|join:", " }}
```

在这种情况下，不能直接使用 `safe` 过滤器，因为它首先将变量转换为字符串，而不是使用序列的各个元素。

片

返回列表的一部分。

使用与Python的列表切片相同的语法。有关介绍，请参见 http://www.diveintopython3.net/native-datatypes.html#slicinglists 。

例：

```
{{ some_list|slice:":2" }}
```

如果 `some_list` 是 ['a'， 'b'， 'c'] >，输出将为 ['a'， 'b'] 。

slugify

转换为ASCII。将空格转换为连字符。删除不是字母数字，下划线或连字符的字符。转换为小写。还剥离前导和尾随空格。

例如：

```
{{ value|slugify }}
```

如果 `value` 是 “Joel 是 a >，输出将为 "joel-is-a-slug" 。

stringformat

根据参数格式化变量，一个字符串格式化说明符。此说明符使用Python字符串格式化语法，除了前导“%”被删除。

有关Python字符串格式的文档，请参见
https://docs.python.org/library/stdtypes.html#string-formatting-operations

例如：

```
{{ value|stringformat:"E" }}
```

如果 value 为 10 ，输出将为 1.000000E+01 。

striptags

尽一切可能努力剥离所有[X] HTML标签。

例如：

```
{{ value|striptags }}
```

If value is
"Joel <button>is</button> a <span&
, the output will be "Joel is a slug" .

无安全保证

请注意，striptags 不会保证其输出是HTML安全的，尤其是对于无效的HTML输入。因此，**NEVER**将 safe 过滤器应用于 striptags 输出。如果您正在寻找更强大的功能，可以使用 bleach Python库，特别是其清洁方法。

时间

根据给定的格式格式化时间。

给定格式可以是预定义的 TIME_FORMAT ，也可以是与 date 过滤器相同的自定义格式。请注意，预定义的格式是与区域设置相关的。

例如：

```
{{ value|time:"H:i" }}
```

如果 value 等效于 datetime.datetime.now() ，则输出将为字符串 "01:23" 。

另一个例子：

假设 USE_L10N 为 True 且 LANGUAGE_CODE 为例如 "de"

```
{{ value|time:"TIME_FORMAT" }}
```

输出将是字符串 `"01:23:00"` （与Django一起提供的 `de` 语言环境的 `"TIME_FORMAT"` 格式说明 `"H:i:s"` ）。

`time` 过滤器只接受格式字符串中与时间相关的参数，而不是日期（由于显而易见的原因）。If you need to format a `date` value, use the `date` filter instead (or along `time` if you need to render a full `datetime` value).

There is one exception the above rule: When passed a `datetime` value with attached timezone information (a *time-zone-aware* `datetime` instance) the `time` filter will accept the timezone-related *format specifiers* `'e'` , `'O'` , `'T'` and `'Z'` .

不使用格式字符串时使用：

```
{{ value|time }}
```

...将使用 `TIME_FORMAT` 设置中定义的格式化字符串，而不应用任何本地化。

Changed in Django 1.7:

在Django 1.7中添加了接收和操作带有时区信息的值的能力。

时光

将日期格式设为自该日期起的时间（例如，“4天，6小时”）。

采用一个可选参数，它是一个包含用作比较点的日期的变量（不带参数，比较点为现在）。例如，如果 `blog_date` 是表示2006年6月1日午夜的日期实例，并且 `comment_date` 是2006年6月1日08:00的日期实例，则以下将返回“8小时”：

```
{{ blog_date|timesince:comment_date }}
```

比较offset-naive和offset-aware datatimes将返回一个空字符串。

分钟是所使用的最小单位，对于相对于比较点的未来的任何日期，将返回“0分钟”。

时间

类似于 `timesince` ，除了它测量从现在开始直到给定日期或日期时间的时间。例如，如果今天是2006年6月1日，而 `conference_date` 是保留2006年6月29日的日期实例，则 `{{ conference_date | timeuntil }}` 将返回“4周”。

使用可选参数，它是一个包含用作比较点的日期（而不是现在）的变量。如果 `from_date` 包含2006年6月22日，则以下内容将返回“1周”：

```
{{ conference_date|timeuntil:from_date }}
```

比较offset-naive和offset-aware datatimes将返回一个空字符串。

分钟是使用的最小单位，对于过去的任何相对于比较点的日期，将返回“0分钟”。

标题

使字符以大写字符开头，其余字符小写，将字符串转换为titlecase。此标记不会努力保持“小写字”小写。

例如：

```
{{ value|title }}
```

如果 `value` 为 `“我 FIRST post”` ，输出将为 `“我的 第一 帖子”` 。

截短体

如果字符串字符多于指定的字符数量，那么会被截断。截断的字符串将以可翻译的省略号序列（“...”）结尾。

参数：要截断的字符数

例如：

```
{{ value|truncatechars:9 }}
```

如果 `value` 是 `“Joel 是 a >，输出将为` “Joel i ...” ` 。`

truncatechars_html

New in Django 1.7.

类似于 `truncatechars` ，除了它知道HTML标记。在字符串中打开并且在截断点之前未关闭的任何标记在截断后立即关闭。

例如：

```
{{ value|truncatechars_html:9 }}
```

If `value` is `"<p>Joel is a slug</p>"`, the output will be `"<p>Joel i...</p>"`.

HTML内容中的换行符将保留。

截断字

在一定数量的字后截断字符串。

参数：要截断的字数

例如：

```
{{ value|truncatewords:2 }}
```

如果 `value` 是 `“Joel 是 a >，输出将是` “Joel 是 ...” 。

字符串中的换行符将被删除。

truncatewords_html

类似于 `truncatewords`，除了它知道HTML标记。在字符串中打开并且在截断点之前未关闭的任何标记在截断后立即关闭。

这比 `truncatewords` 效率较低，因此只应在传递HTML文本时使用。

例如：

```
{{ value|truncatewords_html:2 }}
```

If `value` is `"<p>Joel is a slug</p>"`, the output will be `"<p>Joel is ...</p>"`.

HTML内容中的换行符将保留。

unordered_list

接收一个嵌套的列表，返回一个HTML 的列表 —— 不包含开始和结束的 标签。

列表假设成具有合法的格式。例如，如果 `var` 包含 `['States', ['Kansas', ['Lawrence', 'Topeka'], 'Illinois']]`，那么 `{{ var|unordered_list }}` 将返回：

```
<li>States
<ul>
        <li>Kansas
        <ul>
                <li>Lawrence</li>
                <li>Topeka</li>
        </ul>
        </li>
        <li>Illinois</li>
</ul>
</li>
```

自1.8版起已弃用：还支持旧的，更限制性和详细的输入格式： `['States'， [['Kansas'， t> []]， ['Topeka'， []]]]， t7> [` 。这种语法在Django 2.0 中将不再支持。

上

将字符串转换为大写形式：

例如：

```
{{ value|upper }}
```

如果 `value` 是 `“Joel 是 a >，输出将为` “JOEL IS A SLUG” 。

urlencode

转义要在URL中使用的值。

例如：

```
{{ value|urlencode }}
```

If `value` is `"http://www.example.org/foo?a=b&c=d"` , the output will be `"http%3A//www.example.org/foo%3Fa%3Db%26c%3Dd"` .

可以提供包含不应该转义的字符的可选参数。

如果未提供，则'/'字符被假定为安全的。当所有字符应该转义时，可以提供空字符串。例如：

```
{{ value|urlencode:"" }}
```

如果 `value` 为 `"http://www.example.org/"` ，输出将为 `"http%3A%2F%2Fwww.example.org%2F"` 。

urlize

将文字中的网址和电子邮件地址转换为可点击的链接。

此模板代码适用于以 `http://` ， `https://` 或 `www.` 为前缀的链接。例如， `http://goo.gl/aia1t` 会得到转换，但 `goo.gl/aia1t` 不会。

它还支持以原始顶级域（ `.com` ， `.edu` ， `.gov` ， `.int` ， `.mil` ， `.net` 和 `.org` ）。例如， `djangoproject.com` 被转换。

Changed in Django 1.8:

支持添加包含顶级域名后面的字符（例如， `djangoproject.com/` 和 `djangoproject.com/download/` ）的纯域链接。

链接可以具有结尾标点符号（句点，逗号，近括号）和前导标点符号（开头括号）， `urlize` 仍然可以做正确的事。

由 `urlize` 生成的链接会向其中添加 `rel="nofollow"` 属性。

例如：

```
{{ value|urlize }}
```

如果 `value` 是 `“检查 出 www.djangoproject.com” 将` “检查 出 &lt； a href ="http://www.djangoproject.com" t10> rel ="nofollow"&gt； www.djangoproject.com&lt； / a&gt； “ 。

除了网络链接， `urlize` 还会将电子邮件地址转换为 `mailto:` 链接。如果 `value` 是 `“发送 问题 到 foo@example.com” t6 >` ，输出将是 `“发送 问题 到 &lt； a href =“mailto：foo@example.com”&gt； foo@exam` 。

`urlize` 过滤器还采用可选参数 `autoescape` 。如果 `autoescape` 是 `True` ，则使用Django的内置 escape 过滤器转义链接文字和网址。 `autoescape` 的默认值为 `True` 。

注意

如果 `urlize` 应用于已经包含HTML标记的文本，则会无法正常工作。仅将此过滤器应用于纯文本。

urlizetrunc

将网址和电子邮件地址转换为可点击的链接，就像urlize，但截断长度超过给定字符数限制的网址。

参数：链接文本的字符数应截短为，包括如果截断是必要的，添加的省略号。

例如：

```
{{ value|urlizetrunc:15 }}
```

If `value` is `"Check out www.djangoproject.com"` , the output would be `'Check out <a href="http://www.djangoproject.com" rel="nofollow"&`
.

与urlize一样，此过滤器应仅应用于纯文本。

字数

返回字数。

例如：

```
{{ value|wordcount }}
```

如果 `value` 是 `“Joel 是 a >，输出将为` 4 `。`

wordwrap

以指定的行长度换行单词。

参数：用于换行文本的字符数

例如：

```
{{ value|wordwrap:5 }}
```

如果 `value` 是 `Joel 是 a slug 输出将是：`

```
Joel
is a
slug
```

yesno

将值“ True ， False 和（可选） None 映射到字符串”yes“，”no“，”maybe“自定义映射作为逗号分隔列表传递，并根据值返回其中一个字符串：

例如:

```
{{ value|yesno:"yeah,no,maybe" }}
```

Value	Argument	Outputs
True		yes
True	"yeah,no,maybe"	yeah
False	"yeah,no,maybe"	no
None	"yeah,no,maybe"	maybe
None	"yeah,no"	no (converts None to False if no mapping for None is given)

国际化标签和过滤器

Django提供模板标记和过滤器，以控制模板中*internationalization*的每个方面。它们允许对翻译，格式化和时区转换进行粒度控制。

i18n

此库允许在模板中指定可翻译文本。要启用它，请将 USE_I18N 设置为 True ，然后加载 {% 加载 i18n %} 。

请参阅*Internationalization: in template code*中。

l10n

此库提供对模板中值的本地化的控制。您只需要使用 {% 加载 l10n %} 但您通常会将 USE_L10N 设置为 True ，以便本地化默认处于活动状态。

请参阅*Controlling localization in templates*。

tz

此库提供对模板中时区转换的控制。像 l10n ，您只需要使用 {% 加载 tz } ，但通常还会将 USE_TZ 设置为 True ，以便默认情况下会转换为本地时间。

请参阅*Time zone aware output in templates*。

其他标签和过滤器库

Django附带了一些其他模板标记库，您必须在 `INSTALLED_APPS` 设置中显式启用，并在您的模板中启用 `{% load %}` 标记。

django.contrib.humanize

一组Django模板过滤器，用于向数据添加“人性化”。请参阅*django.contrib.humanize*。

django.contrib.webdesign

在设计网站时有用的模板标签集合，例如Lorem Ipsum文本生成器。请参阅*django.contrib.webdesign*。

静态的

静态的

要链接保存在 `STATIC_ROOT` 中的静态文件，Django附带了 `static` 模板标记。无论是否使用 `RequestContext` ，您都可以使用此方法。

```
{% load static %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

它还能够消耗标准上下文变量，例如。假设将 `user_stylesheet` 变量传递给模板：

```
{% load static %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type=
"text/css" media="screen" />
```

如果您希望在不显示静态网址的情况下检索静态网址，则可以使用略有不同的调用：

```
{% load static %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}"></img>
```

注意

The `staticfiles` contrib app also ships with a `static template tag` which uses `staticfiles'` `STATICFILES_STORAGE` to build the URL of the given path (rather than simply using `urllib.parse.urljoin()` with the `STATIC_URL` setting and the given path). 如果您有高级用例（例如*using a cloud service to serve static files*），请改用此方法：

```
{% load static from staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

get_static_prefix

You should prefer the `static` template tag, but if you need more control over exactly where and how `STATIC_URL` is injected into the template, you can use the `get_static_prefix` template tag:

```
{% load static %}
<img src="{% get_static_prefix %}images/hi.jpg" alt="Hi!" />
```

还有一个第二种形式，你可以使用，以避免额外的处理，如果你需要多次的价值：

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}

<img src="{{ STATIC_PREFIX }}images/hi.jpg" alt="Hi!" />
<img src="{{ STATIC_PREFIX }}images/hi2.jpg" alt="Hello!" />
```

get_media_prefix

类似于`get_static_prefix`，`get_media_prefix`填充媒体前缀为`MEDIA_URL`的模板变量，例如：

```
{% load static %}
<body data-media-url="{% get_media_prefix %}">
```

通过将值存储在数据属性中，如果我们想在JavaScript上下文中使用它，我们确保它适当地转义。

django.contrib.webdesign

自版本1.8后已弃用：程序包仅包含一个模板标记，并已移至内置标记（ `lorem` ）。

django.contrib.humanize

一系列Django的模板过滤器，有助于向数据添加“人文关怀”。

把'django.contrib.humanize'添加到INSTALLED_APPS设置来激活这些过滤器。 执行以上步骤之后，在模板中使用{% load humanize %} ，你就可以访问到下面的过滤器了·。

基数词

对于数字1~9，返回拼写出来的数字。否则返回数字本身。这样遵循了出版的格式。

例如：

- 1 会变成one。
- 2 会变成 two。
- 10 会变成 10。

你可以传递整数，或者整数的字符串形式。

整数间的逗号

将整数转化为字符串，每三位之间带一个逗号。

例如：

- 4500 会变成 4,500。
- 45000 会变成 45,000
- 450000 会变成 450,000。
- 4500000 会变成 4,500,000。

如果启动了格式本地化，将会被遵循。例如，在德语（'de'）中：

- 45000 会变成 '45.000'。
- 450000 会变成 '450.000'。

你可以传递整数，或者整数的字符串形式。

整数词组

将一个大的整数转化为友好的文字表示形式。适用于超过一百万的数字。

例如：

- 1000000 会变成 1.0 million。

- 1200000 会变成 1.2 million。
- 1200000000 会变成 1.2 billion。

支持高达10的100次方 (Googol) 的整数。

如果启动了格式本地化将会被遵循。例如，在德语（'de'）中：

- 1000000 会变成 '1,0 Million'。
- 1200000 会变成 '1,2 Million'。
- 1200000000 会变成 '1,2 Milliarden'。

你可以传递整数，或者整数的字符串形式。

自然日期

对于当天或者一天之内的日期， 返回“今天”，“明天”或者“昨天”，视情况而定。否则，使用传进来的格式字符串给日期格式化。

参数：日期的格式字符串在date标签中描述。

例如（其中“今天”是2007年2月17日）：

- 16 Feb 2007 会变成 yesterday。
- 17 Feb 2007 会变成 today。
- 18 Feb 2007 会变成 tomorrow。

其他日期按照提供的参数格式化，如果没提供参数的话，将会按照DATE_FORMAT设置。

自然时间

对于日期时间的值，返回一个字符串来表示多少秒、分钟或者小时之前 —— 如果超过一天之前，则回退为使用timesince格式。如果是未来的日期时间，返回值会自动使用合适的文字表述。

例如（其中“现在”是2007年2月17日16时30分0秒）：

- 17 Feb 2007 16:30:00 会变成 now。
- 17 Feb 2007 16:29:31 会变成 29 seconds ago。
- 17 Feb 2007 16:29:00 会变成 a minute ago。
- 17 Feb 2007 16:25:35 会变成 4 minutes ago。
- 17 Feb 2007 15:30:29 会变成 59 minutes ago。
- 17 Feb 2007 15:30:01 会变成 59 minutes ago。
- 17 Feb 2007 15:30:00 会变成 an hour ago。
- 17 Feb 2007 13:31:29 会变成 2 hours ago。
- 16 Feb 2007 13:31:29 会变成 1 day, 2 hours ago。
- 16 Feb 2007 13:30:01 会变成 1 day, 2 hours ago。
- 16 Feb 2007 13:30:00 会变成 1 day, 3 hours ago。
- 17 Feb 2007 16:30:30 会变成 30 seconds from now。

- 17 Feb 2007 16:30:29 会变成 29 seconds from now。
- 17 Feb 2007 16:31:00 会变成 a minute from now。
- 17 Feb 2007 16:34:35 会变成 4 minutes from now。
- 17 Feb 2007 17:30:29 会变成 an hour from now。
- 17 Feb 2007 18:31:29 会变成 2 hours from now。
- 18 Feb 2007 16:31:29 会变成 1 day from now。
- 26 Feb 2007 18:31:29 会变成 1 week, 2 days from now。

序数词

将一个整数转化为它的序数词字符串。

例如：

- 1 会变成 1st。
- 2 会变成 2nd。
- 3 会变成 3rd。

你可以传递整数，或者整数的字符串形式。

面向程序员

Django 模板语言：面向Python程序员

本文从技术的角度解释Django 模板系统 —— 它如何工作以及如何继承它。如果你正在查找语言语法方面的参考，参见*Django 模板语言*。

假设你已经理解了模板、上下文、变量、标签和渲染。如果你不熟悉这些概念，从阅读 *Django 模板语言*起步吧。

概述

在Python中使用模板系统有三个步骤：

1. 配置 `引擎` 。
2. 将模板代码编译成 `模板` 。
3. 根据 `上下文` 渲染模板。

对于这些步骤，Django 的项目一般使用*高级的、与后端无关的API* ，而不用模板系统的底层API：

1. Django?对 `TEMPLATES` ?设置中的每个 `DjangoTemplates` 后端实例化一个 `引擎` 。 `DjangoTemplates` 封装 `引擎` 并将它适配成通用的模板后端API。
2. `django.template.loader` 模块提供一些函数例如 `get_template()` 来加载模板。它们返回一个 `django.template.backends.django.Template` ，这个返回值封装了真正的 `django.template.Template` 。
3. 在上一步中获得的 `模板` 有一个 `render()` 方法， 该方法将上下文和HTTP请求添加到 `Context` 中并将渲染委托给底层的 `模板` 。

配置引擎

class `Engine` (*[dirs][, app_dirs][, allowed_include_roots][, context_processors][, debug][, loaders][, string_if_invalid][, file_charset]*)

New in Django 1.8.

实例化 `Engine` 时，所有的参数必须通过关键字参数传递：

- `dirs` 是一个列表，包含引擎查找模板源文件的目录。它用于配置 `filesystem.` Loader。

  默认为一个空的列表。

- `app_dirs` 只影响 `loaders` 的默认值。参见下文。

  默认为 `False` 。

- `allowed_include_roots` 是一个字符串列表，它们表示 `{% ssi %}` 模板标签的前缀。这是一个安全措施，让模板的作者不能访问他们不应该访问的文件。

  例如，如果 `'allowed_include_roots'` 为 `['/home/html', '/var/www']` ，那么 `{% ssi /home/html/foo.txt %}` 可以工作，而 `{% ssi /etc/passwd %}` 不能工作。

  默认为一个空的列表。

```
Deprecated since version 1.8: 废弃`allowed_include_roots`。
```

- `context_processors` 是一个Python 可调用对象的路径列表，它们用于模板渲染时填充其上下文。这些可调用对象接收一个HTTP 请求对象作为它们的参数，并返回一个 字典 用于合并到上下文中。

  它默认为一个空的列表。

  更多信息，参见 `RequestContext` 。

- `debug` 是一个布尔值，表示打开/关闭模板的调试模式。如果为 `True` ，模板引擎将保存额外的调试信息，这些信息可以用来显示模板渲染过程中引发的异常的详细报告。

  它默认为 `False` 。

- `loaders` 是模板加载类的一个列表，这些类由字符串表示。每个 `Loader` 类知道如何从一个特定的源导入模板。还可以使用元组代替字符串表示这些类。元组的第一个元素应该是 `Loader` 类的名称，接下来的元素将在 `Loader` 初始化时用于初始化。

  它默认为包含下面内容的列表：

  - `'django.template.loaders.filesystem.Loader'`
  - `'django.template.loaders.app_directories.` Loader' 当前仅当 `app_dirs` 为 `True` 。

  细节参见*Loader types*。

- `string_if_invalid` 表示模板系统遇到不合法（例如，拼写错误）的变量时应该使用的字符串。

  默认为空字符串。

  细节参见如何处理不合法的变量。

- `file_charset` 是读取磁盘上的模板文件使用的字符集。

  默认为 `'utf-8'` 。

static `Engine.get_default` ()

当Django 项目配置仅配置一个 `DjangoTemplates` 引擎时，这个方法返回底层的 `引擎` 。在其它情况下，它将引发 `ImproperlyConfigured` 。

It's required for preserving APIs that rely on a globally available, implicitly configured engine. Any other use is strongly discouraged.

`Engine.from_string` (*template_code*)

编译给定的template_code 并返回一个 `Template` 对象。

`Engine.get_template` (*template_name*)

根据给定的名称，编译并返回一个 `Template` 对象。

`Engine.select_template` (*self*, *template_name_list*)

类似 `get_template()` ，不同的是它接收一个名称列表并返回找到的第一个模板。

加载模板

建议调用 `Engine` 的工厂方法创建 `Template` ： `get_template()` 、 `select_template()` 和 `from_string()` 。

在 `TEMPLATES` ?只定义一个 `DjangoTemplates` 引擎的Django 项目中，可以直接实例化 `Template` 。

class `Template`

这个类位于 `django.template.Template` 。其构造函数接收一个参数 —— 原始的模板代码：

```
from django.template import Template

template = Template("My name is {{ my_name }}.")
```

幕后

系统只会解析一次原始的模板代码 —— 当创建 `Template` 对象的时候。在此之后，处于性能考虑，会在内部将它存储为一个树形结构。

解析器本身非常快。大部分解析的动作只需要调用一个简短的正则表达式。

渲染上下文

一旦编译好 `Template` 对象，你就可以用上下文渲染它了。你可以使用不同的上下文多次重新渲染相同的模板。

class `Context` (*[dict]*[, current*app]*)

这个类位于 `django.template.Context` 。其构造函数接收两个可选的参数：

- 一个字典，映射变量名到变量的值。
- 当前应用的名称。该应用的名称用于帮助解析带命名空间的*URLs*。如果没有使用带命名空间的URL，可以忽略这个参数。

```
Deprecated since version 1.8: 废弃`current_app` 参数。如果需要它，则必须使用[`RequestContext`](#django.template.RequestContext "django.template.RequestContext") 代替[`Context`](#django.template.Context "django.template.Context")。
```

细节参见下文的使用上下文对象。

`Template.render` (*context*)

使用 `Context` 调用 `Template` 对象的 `render()` 方法来“填充”模板：

```
>>> from django.template import Context, Template
>>> template = Template("My name is {{ my_name }}.")

>>> context = Context({"my_name": "Adrian"})
>>> template.render(context)
"My name is Adrian."

>>> context = Context({"my_name": "Dolores"})
>>> template.render(context)
"My name is Dolores."
```

变量及其查找

变量名必须由字母、数字、下划线（不能以下划线开头）和点组成。

点在模板渲染时有特殊的含义。变量名中点表示查找。具体一点，当模板系统遇到变量名中的一个点时，它会按下面的顺序进行查找：

- 字典查找。例如： `foo["bar"]`
- 属性查找。例如： `foo.bar`
- 列表索引查找。例如： `foo[bar]`

注意，像 `{{ foo.bar }}` 这种模版表达式中的“bar”，如果在模版上下文中存在，将解释为一个字符串字面量而不是使用变量“bar”的值。

模板系统使用找到的第一个可用的类型。这是一个短路逻辑。下面是一些示例：

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ person.first_name }}.")
>>> d = {"person": {"first_name": "Joe", "last_name": "Johnson"}
}
>>> t.render(Context(d))
"My name is Joe."

>>> class PersonClass: pass
>>> p = PersonClass()
>>> p.first_name = "Ron"
>>> p.last_name = "Nasty"
>>> t.render(Context({"person": p}))
"My name is Ron."

>>> t = Template("The first stooge in the list is {{ stooges.0 }
}.")
>>> c = Context({"stooges": ["Larry", "Curly", "Moe"]})
>>> t.render(c)
"The first stooge in the list is Larry."
```

如果变量的任何部分是可调用的，模板系统将尝试调用它。例如：

```
>>> class PersonClass2:
...     def name(self):
...         return "Samantha"
>>> t = Template("My name is {{ person.name }}.")
>>> t.render(Context({"person": PersonClass2}))
"My name is Samantha."
```

可调用的变量比只需直接查找的变量稍微复杂一些。需要记住下面几点：

- 如果变量在调用时引发一个异常，该异常将会传播，除非该异常的 `silent_variable_failure` 属性的值为 `True` 。如果异常确实 具有一个 `silent_variable_failure` 属性且值为 `True` ，该变量将渲染成引擎的 `string_if_invalid` 配置的值（默认为一个空字符串）。例如：

```
>>> t = Template("My name is {{ person.first_name }}.")
    >>> class PersonClass3:
    ...     def first_name(self):
    ...         raise AssertionError("foo")
    >>> p = PersonClass3()
    >>> t.render(Context({"person": p}))
    Traceback (most recent call last):
    ...
    AssertionError: foo

    >>> class SilentAssertionError(Exception):
    ...     silent_variable_failure = True
    >>> class PersonClass4:
    ...     def first_name(self):
    ...         raise SilentAssertionError
    >>> p = PersonClass4()
    >>> t.render(Context({"person": p}))
    "My name is ."
```

```
注意，[`django.core.exceptions.ObjectDoesNotExist`](../exceptions
.html#django.core.exceptions.ObjectDoesNotExist "django.core.exc
eptions.ObjectDoesNotExist") 的`silent_variable_failure = True`，
它是Django 数据库API 所有`DoesNotExist` 异常的基类。所以，如果你在Djang
o 模板中使用Django 模型对象，任何 `DoesNotExist` 异常都将默默地失败。
```

- 只有在变量不需要参数时才可调用。否则，系统将返回引擎的 `string_if_invalid` 选项。
- 很显然，调用某些变量会带来副作用，允许模板系统访问它们将是愚蠢的还会带来安全漏洞。

  每个Django 模型对象的 `delete()` 方法就是一个很好的例子。模板系统不应该允许下面的行为：

```
I will now delete this valuable data. {{ data.delete }}
```

```
设置可调用变量的`alters_data` 属性可以避免这点。如果变量设置`alters_data
=True` ，模板系统将不会调用它，而会无条件使用`string_if_invalid` 替换这
个变量。Django 模型对象自动生成的[`delete()`](../models/instances.ht
ml#django.db.models.Model.delete "django.db.models.Model.delete"
) 和[`save()`](../models/instances.html#django.db.models.Model.s
ave "django.db.models.Model.save") 方法自动 设置`alters_data=True`
。 例如：
```

```
def sensitive_function(self):
        self.database_record.delete()
    sensitive_function.alters_data = True
```

- 有时候，处于某些原因你可能想关闭这个功能，并告诉模板系统无论什么情况下都不要调用变量。设置可调用对象的 `do_not_call_in_templates` 属性的值为 `True` 可以实现这点。模板系统的行为将类似这个变量是不可调用的（例如，你可以访问可调用对象的属性）。

如何处理不合法的变量

一般情况下，如果变量不存在，模板系统会插入引擎 `string_if_invalid` 配置的值，其默认设置为 `''` （空字符串）。

过滤器只有在 `string_if_invalid` 设置为 `''` （空字符串）时才会应用到不合法的变量上。如果 `string_if_invalid` 设置为任何其它的值，将会忽略变量的过滤器。

这个行为对于 `if` 、 `for` 和 `regroup` 模板标签有些不同。如果这些模板便签遇到不合法的变量，会将该变量解释为 `None` 。在这些模板标签中的过滤器对不合法的变量也会始终应用。

如果 `string_if_invalid` 包含 `'%s'` ，这个格式标记将会被非法变量替换。

只用于调试目的！

虽然 `string_if_invalid` 是一个很有用的调试工具，但是，将它作为“默认的开发工具”是个很糟糕的主意。

许多模板包括Admin 站点，在遇到不存在的变量时，依赖模板系统的沉默行为。如果你赋值非 `''` 的值给 `string_if_invalid` ，使用这些模板和站点可能遇到渲染上的问题。

一般情况下，只有在调试一个特定的模板问题时才启用 `string_if_invalid` ，一旦调试完成就要清除。

内置的变量

每个上下文都包含 `True` 、 `False` 和 `None` 。和你期望的一样，这些变量将解析为对应的Python 对象。

字符串字面值的局限

Django 的模板语言没有办法转义它自己的语法用到的字符。例如，如果你需要输出字符序列 `{%` 和 `%}` ，你需要用到 `templatetag` 标签。

如果你想在模板过滤器或标签中包含这些序列，会存在类似的问题。例如，当解析 block 标签时，Django 的模板解析器查找 `%}` 之后出现的第一个 `{%` 。这将导致不能使用 `"%}"` 这个字符串字面量。例如，下面的表达式将引发一个 `TemplateSyntaxError` ：

```
{% include "template.html" tvar="Some string literal with %} in it." %}

{% with tvar="Some string literal with %} in it." %}{% endwith %}
```

在过滤器参数中使用反向的序列会触发同样的问题：

```
{{ some.variable|default:"}}" }}
```

如果你需要使用这些字符串序列，可以将它们保存在模板变量中，或者自定义模板标签或过滤器来绕过这个限制。

使用Context 对象

大部分时候，你将通过传递一个完全填充的字典给 `Context()` 来实例化一个 `Context` 对象。你也可以使用标准的字典语法在 `Context` 对象实例化之后，向它添加和删除元素：

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
Traceback (most recent call last):
...
KeyError: 'foo'
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

`Context.get` (*key*, *otherwise=None*)

如果 `key` 在Context 中，则返回 `key` 的值，否则返回 `otherwise` 。

`Context.pop` ()

`Context.push` ()

exception `ContextPopException`

`Context` 对象是一个栈。也就是说，你可以 `push()` 和 `pop()` 它。如果你 `pop()` 得太多，它将引发 `django.template.ContextPopException` ：

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.push()
{}
>>> c['foo'] = 'second level'
>>> c['foo']
'second level'
>>> c.pop()
{'foo': 'second level'}
>>> c['foo']
'first level'
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'
>>> c.pop()
Traceback (most recent call last):
...
ContextPopException
```

New in Django 1.7.

你还可以使用 `push()` 作为上下文管理器以确保调用对应的 `pop()` 。

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> with c.push():
...     c['foo'] = 'second level'
...     c['foo']
'second level'
>>> c['foo']
'first level'
```

所有传递给 `push()` 的参数将传递给 `dict` 构造函数用于构造新的上下文层级。

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> with c.push(foo='second level'):
...     c['foo']
'second level'
>>> c['foo']
'first level'
```

`Context.update` (*other_dict*)

除了 `push()` 和 `pop()` 之外， `Context` 对象还定义一个 `update()` 方法。它的工作方式类似 `push()` ，不同的是它接收一个字典作为参数并将该字典压入栈。

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.update({'foo': 'updated'})
{'foo': 'updated'}
>>> c['foo']
'updated'
>>> c.pop()
{'foo': 'updated'}
>>> c['foo']
'first level'
```

将 `Context` 用作栈在一些自定义的标签中 非常方便。

`Context.flatten` ()

New in Django 1.7.

利用 `flatten()` 方法，你可以获得字典形式的全部 `Context` 栈，包括内建的变量。

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.update({'bar': 'second level'})
{'bar': 'second level'}
>>> c.flatten()
{'True': True, 'None': None, 'foo': 'first level', 'False': Fals
e, 'bar': 'second level'}
```

`flatten()` 方法在内部还被用来比较 `Context` 对象。

```
>>> c1 = Context()
>>> c1['foo'] = 'first level'
>>> c1['bar'] = 'second level'
>>> c2 = Context()
>>> c2.update({'bar': 'second level', 'foo': 'first level'})
{'foo': 'first level', 'bar': 'second level'}
>>> c1 == c2
True
```

在单元测试中， `flatten()` 的结果可以用于比较 `Context` 和 `dict` ：

```
class ContextTest(unittest.TestCase):
    def test_against_dictionary(self):
        c1 = Context()
        c1['update'] = 'value'
        self.assertEqual(c1.flatten(), {
            'True': True,
            'None': None,
            'False': False,
            'update': 'value',
        })
```

子类化Context：RequestContext

class `RequestContext` (*request[, dict*][, processors]_)

Django 带有一个特殊的 `Context` 类 `django.template.RequestContext`，它与通常的 `django.template.Context` 行为有少许不同。第一个不同点是，它接收一个 `HttpRequest` 作为第一个参数。例如：

```
c = RequestContext(request, {
    'foo': 'bar',
})
```

第二个不同点是，它根据引擎的 `context_processors` 配置选项自动向上下文中填充一些变量。

`context_processors` 选项是一个可调用对象 —— 叫做上下文处理器 —— 的列表，它们接收一个请求对象作为它们的参数并返回需要向上下文中添加的字典。在生成的默认设置文件中，默认的模板引擎包含下面几个上下文处理器：

```
[
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
]
```

Changed in Django 1.8:

Django 1.8 将模板内建的上下文处理器从 `django.core.context_processors` 移动到 `django.template.context_processors` 中。

除此之外， `RequestContext` 始终启用 `'django.template.context_processors.csrf'` 。这是一个安全相关的上下文处理器，Admin 和其它Contrib 应用需要它，而且为了防止意外的错误配置，它被有意硬编码在其中且在 `context_processors` 选项中不可以关闭。

每个处理器按顺序启用。这意味着，如果一个处理器向上下文添加一个变量，而第二个处理器添加一个相同名称的变量，第二个将覆盖第一个。默认的处理器会在下面解释。

上下文处理器应用的时机

上下文处理器应用在上下文数据的顶端。也就是说，上下文处理器可能覆盖你提供给 `Context` 或 `RequestContext` 的变量，所以要注意避免与上下文处理器提供的变量名重复。

如果想要上下文数据的优先级高于上下文处理器，使用下面的模式：

```
from django.template import RequestContext

request_context = RequestContext(request)
request_context.push({"my_name": "Adrian"})
```

Django 通过这种方式允许上下文数据在 `render()` 和 `TemplateResponse` 等 API 中覆盖上下文处理器。

你还可以赋予 `RequestContext` 一个额外的处理器列表，使用第三个可选的位置参数 `processors`。在下面的示例中，`RequestContext` 实例获得一个 `ip_address` 变量：

```
from django.http import HttpResponse
from django.template import RequestContext

def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

def some_view(request):
    # ...
    c = RequestContext(request, {
        'foo': 'bar',
    }, [ip_address_processor])
    return HttpResponse(t.render(c))
```

内建的模板上下文处理器

上下文处理器

下面是每个内置的上下文处理器所做的事情：

django.contrib.auth.context_processors.auth

如果启用这个处理器，每个 `RequestContext` 将包含以下变量：

- `user` — 一个 `auth.` User实例代表当前登录的用户 (或者 一个 `AnonymousUser` 实例, 如果用户没有登录).
- `perms` — 一个 `django.contrib.auth.context_processors.` PermWrapper实例, 代表当前登录用户所拥有的权限.

django.template.context_processors.debug

如果开启这个处理器，每一个 `RequestContext` 将会包含两个变量—但是只有当你的? `DEBUG` 配置设置为 `True` 时有效。请求的IP地址? (`request.` META['REMOTE_ADDR']) is in the `INTERNAL_IPS` setting:

- `debug` — `True` . 你可以在模板中用它测试是否在 `DEBUG` 模式。
- `sql_queries` — ?一个 `{'sql': ..., 'time': ...}` 字典的列表，表示请求期间到目前为止发生的每个SQL 查询及花费的时间。这个列表按查询的顺序排序，并直到访问时才生成。

django.template.context_processors.i18n

如果启用这个处理器，每个 `RequestContext` 将包含两个变量：

- `LANGUAGES` — `LANGUAGES` 设置的值。
- `LANGUAGE_CODE` — `request.` LANGUAGE_CODE, if it exists. 否则为 `LANGUAGE_CODE` 设置的值。

更多信息，参见国际化和本地化。

django.template.context_processors.media

如果启用这个处理器，每个 `RequestContext` 将包含一个 `MEDIA_URL` 变量，表示 `MEDIA_URL` 设置的值。

django.template.context_processors.static

`static` ()[source]

如果启用这个处理器，每个 `RequestContext` 将包含一个 `STATIC_URL` 变量，表示 `STATIC_URL` 设置的值。

django.template.context_processors.csrf

上下文处理器添加一个token，这个token是? `csrf_token` 模版标签需要的，用来针对*Cross Site Request Forgeries*.

django.template.context_processors.request

如果启用这个处理器，每个 `RequestContext` 将包含一个 `request` 变量，表示当前的 `HttpRequest` 。

django.contrib.messages.context_processors.messages

如果启用这个处理器，每个 `RequestContext` 将包含下面两个变量：

- `messages` — 通过消息框架设置的消息（字符串形式）列表。
- `DEFAULT_MESSAGE_LEVELS` — 消息等级名称到它们数值 的映射。

Changed in Django 1.7:

添加 `DEFAULT_MESSAGE_LEVELS` 变量。

编写你自己的上下文处理器

一个上下文处理器有一个非常简单的接口：它是一个参数的Python函数，这个参数是一个 `HttpRequest` 对象，并且返回一个字典，这个字典会被添加到模版上下文中。每个上下文处理器必须返回一个字典。

Custom context processors can live anywhere in your code base. All Django cares about is that your custom context processors are pointed to by the `'context_processors'` option in your `TEMPLATES` setting — or the `context_processors` argument of `Engine` if you're using it directly.

加载模板

通常情况下，你会将模板存储在文件系统上的文件中而不是自己使用底层的 `Template` API。保存模板的目录叫做模板目录。

Django 在许多地方查找模板目录，这取决于你的模板加载设置（参见下文中的"加载器类型"），但是指定模板目录最基本的方法是使用 `DIRS` 选项。

`DIRS`

Changed in Django 1.8:

这个值以前通过 `TEMPLATE_DIRS` 设置定义。

设置文件中 `TEMPLATES` 设置的 `DIRS` 选项或者 `Engine` 的 `dirs` 参数用于告诉Django 你的模板目录。它应该设置为一个字符串列表，包含模板目录的完整路径：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTempla
tes',
        'DIRS': [
            '/home/html/templates/lawrence.com',
            '/home/html/templates/default',
        ],
    },
]
```

模板可以位于任何位置，只要Web 服务器可以读取这些目录和模板。它们可以具有任何扩展名例如 `.html` 或 `.txt` ，或者完全没有扩展名。

注意，这些路径应该使用Unix 风格的前向斜杠，即使在Windows 上。

加载器类型

默认情况下，Django 使用基于文件系统的模板加载器，但是Django 自带几个其它的模板加载器，它们知道如何从其它源加载模板。

默认情况下，某些其它的加载器是被禁用的，但是你可以向 `TEMPLATES` 设置中的 `DjangoTemplates` 添加一个 `'loaders'` 选项或者传递一个 `loaders` 参数给 `Engine` 来激活它们。 `loaders` 应该为一个字符串列表或元组，每个字符串表示一个模板加载器类。下面是Django 自带的模板加载器：

`django.template.loaders.filesystem.Loader`

class `filesystem.Loader`

根据 `DIRS` ，从文件系统加载模板。

该加载器默认是启用的。当然，只有你将 `DIRS` 设置为一个非空的列表，它才能找到模板：

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates'
,
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
}]
```

`django.template.loaders.app_directories.` Loader

class `app_directories.Loader`

从文件系统加载Django 应用中的模板。对于 `INSTALLED_APPS` 中的每个应用，该加载器会查找其下面的一个 `templates` 子目录。如果该目录存在，Django 将在那里查找模板。

这意味着你可以将模板保存在每个单独的应用中。这还使得发布带有默认模板的Django 应用非常方便。

例如，对于这个设置：

```
INSTALLED_APPS = ('myproject.polls', 'myproject.music')
```

... `get_template('foo.html')` 将按顺序在下面的目录中查找 `foo.html` ：

- `/path/to/myproject/polls/templates/`
- `/path/to/myproject/music/templates/`

...并将使用第一个找到的模板。

`INSTALLED_APPS` 的顺序非常重要！例如，如果你想自定义Django Admin，你可能选择使用 `myproject.polls` 中自己的 `admin/base_site.html` 覆盖 `django.contrib.admin` 中标准的 `admin/base_site.html` 。那么你必须确保在 `INSTALLED_APPS` 中 `myproject.polls` 位于 `django.contrib.admin` 之前，否则仍将加载 `django.contrib.admin` 中的模板并忽略你自己的模板。

注意，加载器在第一次运行时会做一些优化：它缓存一个含具有 `templates` 子目录的 `INSTALLED_APPS` 包的列表。

你可以简单地通过设置 `APP_DIRS` 为 `True` 来启用这个加载器：

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates'
,
    'APP_DIRS': True,
}]
```

`django.template.loaders.eggs.Loader`

class `eggs.Loader`

与上面的 `app_directories` 类似，只是它从Python eggs 中而不是从文件系统中加载模板。

这个加载器在默认情况下是禁用的。

`django.template.loaders.cached.Loader`

class `cached.Loader`

默认情况下，每当模版需要被渲染，模版系统将会读取和编译你的模版。但是，Django模版系统是非常高速的，?the overhead from reading and compiling? templates can add up.

基于缓存的模版加载器是一个基于类的加载器，that you configure with a list of other loaders?that it should wrap.The wrapped loaders are used to locate unknown templates when they are first encountered. 接下来，基于缓存加载器将编译过的 `Template` 存储在内存中。The cached `Template` instance is returned for subsequent requests to load the same template.

For example, to enable template caching with the `filesystem` and `app_directories` template loaders you might use the following settings:

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates'
,
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    'OPTIONS': {
        'loaders': [
            ('django.template.loaders.cached.Loader', [
                'django.template.loaders.filesystem.Loader',
                'django.template.loaders.app_directories.Loader'
,
            ]),
        ],
    },
}]
```

Note

All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or that you wrote yourself, you should ensure that the `Node` implementation for each tag is thread-safe. For more information, see *template tag thread safety considerations*.

这个加载器在默认情况下是禁用的。

`django.template.loaders.locmem.Loader`

New in Django 1.8.

class `locmem.Loader`

从一个Python 目录中加载模板。它主要用于测试。

这个加载器接收一个模板字典作为它的第一个参数：

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates'
,
    'OPTIONS': {
        'loaders': [
            ('django.template.loaders.locmem.Loader', {
                'index.html': 'content here',
            }),
        ],
    },
}]
```

这个加载器在默认情况下是禁用的。

Django 按照 `'loaders'` 中选项的顺序使用模板加载器。它逐个使用每个加载器直到某个加载器找到一个匹配的模板。

自定义加载器

自定义 `加载器` 应该继承 `django.template.loaders.base.Loader` 并覆盖 `load_template_source()` 方法，这个方法接收一个 `template_name` 参数、从磁盘（或其它地方）加载模板、然后返回一个元组： `(template_string, template_origin)` 。

Changed in Django 1.8:

`django.template.loaders.base.Loader` 以前定义在 `django.template.loader.BaseLoader` 中。

`Loader` 类的 `load_template()` 方法通过调用 `load_template_source()` 获取模板字符串、从模板源中实例化一个 `Template` 、然后返回一个元组： `(template, template_origin)` 。

模板的**origin** 属性

New in Django 1.7.

当 `Engine` 使用 `debug=True` 初始化时，它的模板将具有一个 `origin` 属性，其值取决于模板加载的源。对于Django 初始化的引擎， `debug` 默认为 `DEBUG` 设置的值。

class `loader.LoaderOrigin`

从模板加载器创建的模板将使用 `django.template.loader.LoaderOrigin` 类。

`name`

模板加载器返回的模板路径。对于从文件系统读取模板的加载器，它为模板的完整路径。

`loadname`

传递给模板加载器的模板相对路径。

class `StringOrigin`

从 `Template` 类创建的模板将使用 `django.template.StringOrigin` 类。

`source`

用于创建模板的字符串。

自定义模板标签和过滤器

为了解决应用中展示逻辑的需求，Django的模板语言提供了各式各样的内建标签以及过滤器。然而，你或许会发现模板内建的这些工具集合不一定能全部满足你的功能需要。在Python中，你可以通过自定义标签或过滤器的方式扩展模板引擎的功能，并使用 `{{ load }}` 标签在你的模板中进行调用。

代码布局

自定义模板标签和过滤器必须位于Django 的某个应用中。如果它们与某个已存在的应用相关，那么将其与应用绑在一起才有意义；否则，就应该创建一个新的应用来包含它。

这个应用应该包含一个 `templatetags` 目录，和 `models.py` 、 `views.py` 等文件处于同一级别目录下。如果目录不存在则创建它——不要忘记创建 `__init__.py` 文件以使得该目录可以作为Python 的包。在添加这个模块以后，在模板里使用标签或过滤器之前你将需要重启服务器。

你的自定义的标签和过滤器将放在 `templatetags` 目录下的一个模块里。这个模块的名字是你稍后将要载入标签时使用的，所以要谨慎的选择名字以防与其他应用下的自定义标签和过滤器名字冲突。

例如，你的自定义标签/过滤器在一个名为 `poll_extras.py` 的文件中，那么你的app目录结构看起来应该是这样的：

```
polls/
    __init__.py
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

然后你可以在模板中像如下这样使用：

```
{% load poll_extras %}
```

为了让 `{{ load }}` 标签工作，包含自定义标签的应用必须在 `INSTALLED_APPS` 中。这是一种安全功能：它允许你在单个主机上Host 许多模板库的Python 代码，而不必让每个Django 都可以访问所有的模板库。

在 `templatetags` 包中放多少个模块没有限制。只需要记住 `{% load %}` 声明将会载入给定模块名中的标签/过滤器，而不是应用的名称。

为了成为一个可用的标签库，这个模块必须包含一个名为 `register` 的变量，它是 `template.Library` 的一个实例，所有的标签和过滤器都是在其中注册的。所以把如下的内容放在你的模块的顶部：

```
from django import template

register = template.Library()
```

幕后

对于大量的示例，请阅读Django的默认过滤器和标记的源代码。它们分别位于 `django/template/defaultfilters.py` 和 `django/template/defaulttags.py` 中。

有关 `load` 标签的更多信息，请阅读其文档。

编写自定义模板过滤器

自定义过滤器就是一个带有一个或两个参数的Python 函数：

- （输入的）变量的值 —— 不一定是字符串形式。
- 参数的值 —— 可以有一个初始值，或者完全不要这个参数。

例如，在 `{{ var|foo:"bar" }}` 中，`foo` 过滤器应当传入变量 `var` 和参数 `"bar"` 。

由于模板语言没有提供异常处理，任何从过滤器中抛出的异常都将会显示为服务器错误。因此，如果有合理的值可以返回，过滤器应该避免抛出异常。在模板中有一个明显错误的情况下，引发一个异常可能仍然要好于用静默的失败来掩盖错误。

这是一个定义过滤器的例子：

```
def cut(value, arg):
    """Removes all values of arg from the given string"""
    return value.replace(arg, '')
```

下面是这个过滤器应该如何使用：

```
{{ somevariable|cut:"0" }}
```

大多数过滤器没有参数。在这种情况下，你的函数不带这个参数即可。示例：

```
def lower(value): # Only one argument.
    """Converts a string into all lowercase"""
    return value.lower()
```

注册自定义过滤器

`django.template.Library.``filter` ()

一旦你写好了你的自定义过滤器函数，你就开始需要把它注册为你的 `Library` 实例，来让它在Django模板语言中可用：

```
register.filter('cut', cut)
register.filter('lower', lower)
```

`Library.filter()` 方法需要两个参数：

1. 过滤器的名称（一个字符串对象）
2. 编译的函数 – 一个Python函数（不要把函数名写成字符串）

你还可以把 `register.filter()` 用作装饰器：

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()
```

如果你像上面第二个例子一样没有声明 `name` 参数，Django将使用函数名作为过滤器的名字。

最后， `register.filter()` 还接收三个关键字参数， `is_safe` 、 `needs_autoescape` 和 `expects_localtime` 。这些参数将在下边过滤器和自动转义 以及过滤器和时区 章节中介绍。

期望字符串的过滤器

`django.template.defaultfilters.``stringfilter` ()

如果你正在编写一个只希望用一个字符串来作为第一个参数的模板过滤器，你应当使用 `stringfilter` 装饰器。这将在对象被传入你的函数之前把这个对象转换成它的字符串值：

```
from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
@stringfilter
def lower(value):
    return value.lower()
```

用这种方式，你甚至可以给这个过滤器传递一个整数，并且不会出现 `AttributeError` (因为整数没有 `lower()` 方法).

过滤器和自动转义

编写一个自定义的过滤器时，请考虑一下过滤器如何与Django 的自定转义行为相互作用。请注意有三种类型的字符串可以传递给模板中的代码：

- 原始字符串 即Python 原生的 `str` 或 `unicode` 类型。输出时，如果自动转义生效则进行转义，否则保持不变。
- 安全字符串 是指在输出时已经被标记为安全而不用进一步转义的字符串。任何必要的转义已经完成。它们通常用于包含HTML 的输出，并希望在客户端解释为原始的形式。

  在内部，这些字符串是 `SafeBytes` 或 `SafeText` 类型。它们共享一个公共基类 `SafeData`，所以你可以使用类似的代码测试他们：

  ```
  if isinstance(value, SafeData):
      # Do something with the "safe" string.
      ...
  ```

- 标记为“需要转义”的字符串 在输出时始终转义，无论它们是否在 `autoescape` 块。然而，即使已经应用自动转义，这些字符也只会转义一次。

  在内部，这些字符串是 `EscapeBytes` 或 `EscapeText` 类型。通常你不需要担心这些；它们用于 `escape` 过滤器的实现。

模板过滤代码最终是这两种中的一个：

1. 你的过滤器没有引进任何HTML 不安全字符（ < 、 > 、 ' 、 " 或 & ）到结果中。在这种情况下，你可以让Django 照顾你的所有的自动转义处理。你需要做的就是当你注册过滤器函数的时候将 `is_safe` 标志设置为 `True`，如下所示：

```
@register.filter(is_safe=True)
def myfilter(value):
    return value
```

这个标志告诉Django 如果"安全"的字符串传递到您的筛选器，结果仍将是"安全"，如果一个非安全字符串传递，如果必要Django 会自动转义它。

你可以认为这个的意思是"此过滤器是安全的 —— 它没有引入任何不安全的 HTML 的可能性。

`is_safe` 存在的必要原因是因为有很多正常的字符串操作会将一个 `SafeData` 对象转换回正常的 `str` 或 `unicode` 对象而不是试图捕获它们，Django 在过滤器完成之后会修复这种破坏。

例如，假设你有一个过滤器将字符串 `xx` 添加到任何输入的末尾。因为这没有引入危险的HTML 字符（已经存在的除外）的结果，你应该使用 `is_safe` 标记你的过滤器：

```
@register.filter(is_safe=True)
def add_xx(value):
    return '%sxx' % value
```

当这个过滤器用在模板中启用自动转义的地方时，如果输入没有标记为“安全”，Django 将对输出进行转义。

默认情况下， `is_safe` 为 `False` ，你可以在不需要的任何过滤器中省略它。

决定你的过滤器是否真的会保持安全字符串是安全的时要小心。如果你在删除字符，可能会无意中在结果留下不平衡的 HTML 标记或实体。例如，从输入删除 `>` 可能将 `<a>` 转变成 `<a` ，这将需要对输出进行转义，避免造成问题。同样，删除一个分号（ `;` ）可以将 `&` 转变成 `&amp` ，不再是一个有效的实体，因此需要进一步的转义。大多数情况下不会这么棘手，但在审查你的代码时要留意任何类似的问题。

标记过滤器位 `is_safe` 将强制过滤器的返回值为字符串。如果你的过滤器应返回一个布尔值或其他非字符串值，则将其标记 `is_safe` 会有意想不到的后果（如将布尔值 False 转换为字符串 'False'）。

2. 或者，你的过滤器代码手动照顾任何必要的转义。这在你正引入新的HTML 标记到结果中时是必要的。你想标记输出为安全的而不用进一步的转义，所以你需要自己处理输入输出。

   用 `django.utils.safestring.mark_safe()` 标记输出为安全字符。

   但你要小心。你需要做的不仅仅只是标记作为安全输出。您需要确保它真的是安全的，而你做什么取决于自动转义是否有效。这个想法的目的是编写的过滤器在无论模板自动转义是打开或关闭时都可以工作，这样模板作者使用起来更

简单。

为了使你的过滤器知道当前的自动转义状态，当你注册过滤器函数时需要设置 `needs_autoescape` 标志为 `True` 。（如果不指定此标志，则默认为 `False` ）。此标志告诉Django 你的过滤器函数想要被传递一个额外的关键字参数，称为 `autoescape` ，如果启用自动转义则为 `True` ，否则为 `False` 。建议设置 `autoescape` 参数的默认值设置为 `True` ，这样如果从Python 代码中调用该函数则会自动启用转义。

例如，让我们编写一个强调字符串第一个字符的过滤器：

```
from django import template
from django.utils.html import conditional_escape
from django.utils.safestring import mark_safe

register = template.Library()

@register.filter(needs_autoescape=True)
def initial_letter_filter(text, autoescape=True):
    first, other = text[0], text[1:]
    if autoescape:
        esc = conditional_escape
    else:
        esc = lambda x: x
    result = '&lt;strong&gt;%s&lt;/strong&gt;%s' % (esc(firs
t), esc(other))
    return mark_safe(result)
```

`needs_autoescape` 标志和 `autoescape` 关键字参数意味着我们的函数将知道当调用过滤器时自动转义是否有效。我们使用 `autoescape` 来决定是否需要通过 `django.utils.html.conditional_escape` 传递输入数据。（在后一种情况下，我们只使用identity函数作为“escape”函数。） `conditional_escape()` 函数与 `escape()` 类似，只不过它只转义了而不是 a `SafeData` 如果将 `SafeData` 实例传递给 `conditional_escape()` ，则数据不会改变。

最后，在上面的例子中，我们记得将结果标记为安全的，这样我们的HTML直接插入到模板中，而不需要进一步转义。

在这种情况下，没有必要担心 `is_safe` 标志（虽然包括它不会伤害任何东西）。每当你手动处理自动转义问题并返回一个安全的字符串， `is_safe` 标志不会改变任何方式。

警告

在重用内置过滤器时避免XSS漏洞

Changed in Django 1.8.

Django的内置过滤器默认情况下具有 `autoescape=True` ，以便获得正确的自动转义行为并避免跨站点脚本漏洞。

在旧版本的Django中，在重新使用Django的内置过滤器时请小心，因为 `autoescape` 默认为 `None` 。您需要传递 `autoescape=True` 才能获得自动转义。

例如，如果您想编写一个称为 `urlize_and_linebreaks` 的自定义过滤器，它结合了 `urlize` 和 `linebreaksbr` 过滤器，过滤器将如下所示：

```
from django.template.defaultfilters import linebreaksbr, urlize

@register.filter(needs_autoescape=True)
def urlize_and_linebreaks(text, autoescape=True):
    return linebreaksbr(
        urlize(text, autoescape=autoescape),
        autoescape=autoescape
    )
```

然后：

```
{{ comment|urlize_and_linebreaks }}
```

将等同于：

```
{{ comment|urlize|linebreaksbr }}
```

过滤器和时区

如果你在编写操作 `datetime` 对象的自定义过滤器，你注册时通常需要将 `expects_localtime` 标志设置为 `True` ：

```
@register.filter(expects_localtime=True)
def businesshours(value):
    try:
        return 9 <= value.hour < 17
    except AttributeError:
        return ''
```

当设置了此标志，如果你的过滤器的第一个参数是时区相关的日期时间值，Django会根据模板中的时区转换规则 在把它传递到你的过滤器之前将其转换为当前时区。

编写自定义的模板标签

标签比过滤器更复杂，因为标签可以做任何事情。Django 提供大量的快捷方式，使编写大多数类型的标签更容易。首先我们要探讨这些快捷方式，然后再解释当快捷方式不够用时如何为这些情况从头开始编写标签。

简单的标签

`django.template.Library.``simple_tag` ()

许多模板标签接收多个参数 —— 字符串或模板变量 —— 并在基于输入的参数和一些其它外部信息进行一些处理后返回一个字符串。例如，`current_time` 标签可能接受一个格式字符串，并返回与之对应的格式化后的时间。

为了简单化这些类型标签的创建，Django 提供一个辅助函数 `simple_tag` 。这个函数是 `django.template.Library` 的一个方法，接受一个任意数目的参数的函数，将其包装在一个 `render` 函数和上面提到的其他必要位，并在模板系统中注册它。

我们的 `current_time` 函数从而可以这样写：

```
import datetime
from django import template

register = template.Library()

@register.simple_tag
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

关于 `simple_tag` 辅助函数几件值得注意的事项：

- 检查所需参数的数量等等，在我们的函数调用的时刻已经完成，所以我们不需要做了。
- 参数（如果有）的引号都已经被截掉，所以我们收到的只是一个普通字符串。
- 如果该参数是一个模板变量，传递给我们的函数是当前变量的值，不是变量本身。

如果你的模板标签需要访问当前上下文，你可以在注册标签时使用 `takes_context` 参数：

```
@register.simple_tag(takes_context=True)
def current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)
```

请注意，第一个参数必须称作 `context` 。

`takes_context` 选项的工作方式的详细信息，请参阅包含标签。

如果你需要重命名你的标签，你可以给它提供自定义的名称：

```
register.simple_tag(lambda x: x - 1, name='minusone')

@register.simple_tag(name='minustwo')
def some_function(value):
    return value - 2
```

`simple_tag` 函数可以接受任意数量的位置参数和关键字参数。例如：

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

然后在模板中，可以将任意数量的由空格分隔的参数传递给模板标签。像在Python中一样，关键字参数的值的设置使用等号（" `=` "），并且必须在位置参数之后提供。例如：

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=us
er.profile %}
```

Inclusion 标签

`django.template.Library.``inclusion_tag` ()

另一种常见类型的模板标签是通过渲染另外一个模板来显示一些数据。例如，Django 的Admin 界面使用自定义模板标签显示"添加/更改"表单页面底部的按钮。这些按钮看起来总是相同，但链接的目标根据正在编辑的对象而变化 —— 所以它们是使用小模板展示当前对象详细信息很好的例子。（在Admin 界面这种情况下，它是 `submit_row` 标记）。

这些类型的标签被称为"Inclusion 标签"。

示例最能体现如何编写Inclusion 标签。让我们编写一个根据给定的教程中创建的 `Poll` 对象输出一个选项列表的标签。标签的用法像这样：

```
{% show_results poll %}
```

... 输出将像这样：

```
<ul>
  <li>First choice</li>
  <li>Second choice</li>
  <li>Third choice</li>
</ul>
```

首先，定义接收这个参数并产生数据字典作为结果的函数。这里重要的一点是，我们只需要返回一个字典，不需要任何复杂的东西。它将用做模板片段的模板上下文。例如：

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

接下来，创建用于渲染标签输出的模板。这个模板是标签固定的功能：标签的编写者指定它，不是模板设计者。在我们的例子中，模板非常简单：

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

现在，通过调用 `Library` 对象的 `inclusion_tag()` 方法创建并注册Inclusion 标签。在我们的示例中，如果上面的模板叫做 `results.html` 文件，并位于模板加载程序搜索的目录，我们将这样注册标签：

```
# Here, register is a django.template.Library instance, as befor
e
@register.inclusion_tag('results.html')
def show_results(poll):
    ...
```

或者可以使用 `django.template.Template` 实例注册Inclusion标签：

```
from django.template.loader import get_template
t = get_template('results.html')
register.inclusion_tag(t)(show_results)
```

......当首次创建该函数时。

有时，你的Inclusion 标签可能要求一大堆参数，这让模板作者非常痛苦，因为不仅要传递这些参数还要记住它们的顺序。为了解决这个问题， Django 提供了一个 `takes_context` 选项给Inclusion 标签。如果你在创建模板标签时指

定 `takes_context` ，这个标签将不需要必选参数，当标签被调用的时候底层的 Python 函数将有一个参数 —— 模板上下文。

比如说,当你想要写一个 inclusion tag总是应用在上下文中，包含 `home_link` 和 `home_title` 这两个用来返回主页的变量。 如下所示:

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

注意函数的第一个参数必须叫做 `context` 。

在 `register.inclusion_tag()` 这一行，我们指定了 `takes_context=True` 和模板的名字。这里是模板 `link.html` 看起来的样子：

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

然后，当任何时候你想调用这个自定义的标签， load 它的 library 然后不需要任何参数就是调用它，就像这样：

```
{% jump_link %}
```

注意当你使用 `takes_context=True` ，就不需要传递参数给这个模板标签。它会自己去获取上下文。

`takes_context` 参数默认为 `False` 。当它设置为 `True` 时，会传递上下文对象给这个标签，如本示例所示。这是这个示例和前面的 `inclusion_tag` 示例的唯一区别。

`inclusion_tag` 函数可以接受任意数量的位置参数和关键字参数。例如：

```
@register.inclusion_tag('my_template.html')
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

然后在模板中，可以将任意数量的由空格分隔的参数传递给模板标签。像在Python 中一样，关键字参数的值的设置使用等号（" `=` "） ，并且必须在位置参数之后提供。例子：

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=us
er.profile %}
```

赋值标签

`django.template.Library.``assignment_tag` ()

为了简单化设置上下文中变量的标签的创建，Django 提供一个辅助函数 `assignment_tag` 。这个函数方式的工作方式与*simple_tag* 相同，不同之处在于它将标签的结果存储在指定的上下文变量中而不是直接将其输出。

我们之前的 `current_time` 函数从而可以这样写：

```
@register.assignment_tag
def get_current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

然后你可以使用 `as` 参数后面跟随变量的名称将结果存储在模板变量中，并将它输出到你觉得合适的地方：

```
{% get_current_time "%Y-%m-%d %I:%M %p" as the_time %}
<p>The time is {{ the_time }}.</p>
```

如果你的模板标签需要访问当前上下文，你可以在注册标签时使用 `takes_context` 参数：

```
@register.assignment_tag(takes_context=True)
def get_current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)
```

注意函数的第一个参数必须叫做 `context` 。

`takes_context` 选项的工作方式的详细信息，请参阅包含标签。

`assignment_tag` 函数可以接受任意数量的位置参数和关键字参数。例如：

```
@register.assignment_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

然后在模板中，可以将任意数量的由空格分隔的参数传递给模板标签。像在Python中一样，关键字参数的值的设置使用等号（" `=` "），并且必须在位置参数之后提供。例子：

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile as the_result %}
```

自定义模板标签进阶

有时创建自定义模板标签的基本功能是不够的。别担心，Django 给你建立模板标签所需的从底层访问完整的内部。

概述

模板系统的运行分为两步：编译和渲染。若要定义一个自定义的模板标签，你指定编译如何工作以及渲染如何工作。

当Django 编译一个模板时，它将原始模板文本拆分成节点。每个节点是 `django.template.Node` 的一个实例，并且有一个 `render()` 方法。编译后的模板就是一个简单 `Node` 对象的列表。当你在编译后的模板对象上调用 `render()` 时，该模板将结合给定的上下文调用每个 `Node` 的 `render()` 。结果所有串联在一起形成该模板的输出。

因此，若要定义一个自定义的模板标签，你需要指定原始模板标签如何被转换成一个 `Node(节点)` （编译函数），以及该节点的 `render()` 方法会进行的渲染动作

写编译函数

解析器处理每个模板标签时，会调用标签上下文对应的函数和对象本身。这个函数会会返回一个 `Node` 实例

例如，让我们写一个简单的模板标签的完整实现： `}} current_time }}` 根据 `strftime()` 语法中标记中给出的参数格式化的当前日期/时间。在任何事情之前决定标记语法是个好主意。在我们的例子中，我们假设标签应该像这样使用：

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

此函数的解析器应抓取参数并创建 `节点` 对象：

```
from django import template

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError(
            "%r tag requires a single argument" % token.contents
.split()[0]
        )
    if not (format_string[0] == format_string[-1] and format_str
ing[0] in ('"', "'")):
        raise template.TemplateSyntaxError(
            "%r tag's argument should be in quotes" % tag_name
        )
    return CurrentTimeNode(format_string[1:-1])
```

笔记：

- `parser` 是模板解析器对象。在这个例子中我们不需要它。
- `token.contents` 是标记的原始内容的字符串。在我们的示例中，它是 `'current_time “%Y-%m-%d %I:%M %p “'` ”。
- `token.split_contents()` 方法将空格上的参数分隔开，同时将带引号的字符串保存在一起。更直接的 `token.contents.split()` 将不会那么健壮，因为它会在所有空间（包括引用字符串中的那些）上天真地分割。始终使用 `token.split_contents()` 是一个好主意。
- 此函数负责提高 `django.template.TemplateSyntaxError` ，包含有用的消息，任何语法错误。
- `TemplateSyntaxError` 异常使用 `tag_name` 变量。不要在错误消息中硬编码标记的名称，因为它会将标记的名称与您的函数相关联。 `token.contents.split()[0]` 将“永远”是您的标记的名称 - 即使标记没有参数。
- 该函数返回 `CurrentTimeNode` ，其中包含节点需要了解的关于此标记的所有内容。在这种情况下，它只传递参数 - “%Y-%m-%d %I:%M %p” t3 > 。模板标记的前导和尾部引号在 `format_string[1:-1]` 中删除。
- 解析是非常低级的。Django开发人员已经尝试在这个解析系统之上编写小框架，使用诸如EBNF语法的技术，但是这些实验使得模板引擎太慢。它是低级的，因为这是最快的。

写入渲染器

编写自定义标记的第二步是定义具有 `render()` 方法的 `Node` 子类。

继续上面的例子，我们需要定义 `CurrentTimeNode` ：

```
import datetime
from django import template

class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        return datetime.datetime.now().strftime(self.format_stri
ng)
```

笔记：

- `__init__()` 从 `do_current_time()` 获取 `format_string` 。始终通过其 `__init__()` 将任何选项/参数/参数传递到 `Node` 。
- `render()` 方法是工作实际发生的地方。
- `render()` 通常应该默认失败，特别是在生产环境中。但在某些情况下，特别是 `context.template.engine.debug` 是 `True` 时，此方法可能会引发异常，使调试更容易。例如，如果几个核心标记接收到错误的数字或类型的参数，则会产生 `django.template.TemplateSyntaxError` 。

最终，编译和渲染的这种解耦导致了高效的模板系统，因为模板可以呈现多个上下文而不必被多次解析。

自动转义注意事项

模板标签的输出不是自动运行通过自动转义过滤器。但是，在编写模板标记时，您仍然需要记住一些事情。

如果模板的 `render()` 函数将结果存储在上下文变量中（而不是返回字符串中的结果），则应小心调用 `mark_safe()` 当变量最终呈现时，它会受到当时有效的自动转义设置的影响，因此应该避免进一步转义的内容需要标记为这样。

此外，如果您的模板标记为执行某些子呈现创建了一个新的上下文，请将auto-escape属性设置为当前上下文的值。 `Context` 类的 `__init__` 方法使用一个名为 `autoescape` 的参数，可以用于此目的。例如：

```
from django.template import Context

def render(self, context):
    # ...
    new_context = Context({'var': obj}, autoescape=context.autoe
scape)
    # ... Do something with new_context ...
```

这不是一个很常见的情况，但它是有用的，如果你自己渲染一个模板。例如：

```
def render(self, context):
    t = context.template.engine.get_template('small_fragment.htm
l')
    return t.render(Context({'var': obj}, autoescape=context.aut
oescape))
```

Changed in Django 1.8:

在Django 1.8中添加了 `Context` 对象的 `template` 属性。 `context.template.engine.get_template` must be used instead of `django.template.loader.get_template()` because the latter now returns a wrapper whose `render` method doesn't accept a `Context` .

如果我们在此示例中忽略将当前的 `context.autoescape` 值传递给我们的新 `Context` ，则结果将自动转义 ，如果在 `}} autoescape off }}` 块。

线程安全注意事项

一旦节点被解析，其 `render` 方法可以被调用任何次数。由于Django有时在多线程环境中运行，因此单个节点可以响应于两个单独的请求而用不同的上下文同时呈现。因此，确保您的模板标记是线程安全的很重要。

为了确保你的模板标签是线程安全的，你不应该在节点本身存储状态信息。例如，Django提供了一个内置的 `cycle` 模板标签，每次呈现时都会在给定字符串列表之间循环：

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

`CycleNode` 的简单实现可能如下所示：

```
import itertools
from django import template

class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cycle_iter = itertools.cycle(cyclevars)

    def render(self, context):
        return next(self.cycle_iter)
```

但是，假设我们有两个模板同时从上面呈现模板片段：

1. 线程1执行其第一次循环迭代， `CycleNode.render()` 返回'row1'
2. 线程2执行其第一次循环迭代， `CycleNode.render()` 返回'row2'
3. 线程1执行其第二次循环迭代， `CycleNode.render()` 返回'row1'
4. 线程2执行其第二循环迭代， `CycleNode.render()` 返回'row2'

CycleNode是迭代，但它是全局迭代。就线程1和线程2而言，它总是返回相同的值。这显然不是我们想要的！

为了解决这个问题，Django提供了与当前正在渲染的模板的 `context` 相关联的 `render_context` 。 `render_context` 的行为类似于Python字典，应该用于在 `render` 方法的调用之间存储 `Node` 状态。

让我们重构我们的 `CycleNode` 实现以使用 `render_context` ：

```
class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cyclevars = cyclevars

    def render(self, context):
        if self not in context.render_context:
            context.render_context[self] = itertools.cycle(self.
cyclevars)
        cycle_iter = context.render_context[self]
        return next(cycle_iter)
```

请注意，存储在 `Node` 的整个生命周期中不会改变的全局信息作为属性是完全安全的。在 `CycleNode` 的情况下， `cyclevars` 参数在实例化 `Node` 后不会改变，因此我们不需要将其 `render_context` 。但是，当前正在渲染的模板特定的状态信息（如 `CycleNode` 的当前迭代）应存储在 `render_context` 中。

注意

请注意我们如何使用 `self` 来限定 `render_context` 内的 `CycleNode` 特定信息。在给定模板中可能有多个 `CycleNodes` ，因此我们需要注意不要破坏另一个节点的状态信息。执行此操作的最简单方法是始终使用 `self` 作为 `render_context` 中的键。如果你跟踪几个状态变量，使 `render_context[self]` 一个字典。

注册标签

最后，按照上述*writing custom template filters*中的说明，向模块的 `Library` 实例注册代码。例：

```
register.tag('current_time', do_current_time)
```

`tag()` 方法有两个参数：

1. 模板标记的名称 - 字符串。如果省略，将使用编译函数的名称。
2. 编译函数 - 一个Python函数（不是作为字符串的函数的名称）。

与过滤器注册一样，也可以将其用作装饰器：

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    ...

@register.tag
def shout(parser, token):
    ...
```

如果您省略 `name` 参数，如上面的第二个示例，Django将使用函数的名称作为标记名称。

将模板变量传递给标记

虽然您可以使用 `token.split_contents()` 向模板标记传递任意数量的参数，但所有参数都会作为字符串文字解压缩。为了将动态内容（模板变量）传递给模板标签作为参数，需要更多的工作。

虽然先前的示例已将当前时间格式化为字符串并返回了字符串，但假设您希望从对象传递 `DateTimeField` ，并具有date-time的模板标记格式：

```
<p>This post was last updated at {% format_time blog_entry.date_
updated "%Y-%m-%d %I:%M %p" %}.</p>
```

最初， `token.split_contents()` 将返回三个值：

1. 标记名称 `format_time` 。
2. 字符串 `'blog_entry.date_updated'` （不带周围的引号）。
3. 格式字符串 `'“%Y-%m-%d %I：%M %p”'` 。``split_contents()`的返回值将包括字符串文字的前导和尾部引号，如下所示。

现在您的标记应该开始看起来像这样：

```
from django import template

def do_format_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, date_to_be_formatted, format_string = token.sp
lit_contents()
    except ValueError:
        raise template.TemplateSyntaxError(
            "%r tag requires exactly two arguments" % token.cont
ents.split()[0]
        )
    if not (format_string[0] == format_string[-1] and format_str
ing[0] in ('"', "'")):
        raise template.TemplateSyntaxError(
            "%r tag's argument should be in quotes" % tag_name
        )
    return FormatTimeNode(date_to_be_formatted, format_string[1:
-1])
```

您还必须更改渲染器以检索 `blog_entry` 对象的 `date_updated` 属性的实际内容。这可以通过使用 `django.template` 中的 `Variable()` 类来实现。

要使用 `Variable` 类，只需使用要解析的变量名称实例化它，然后调用 `variable.resolve(context)` 。所以，例如：

```
class FormatTimeNode(template.Node):
    def __init__(self, date_to_be_formatted, format_string):
        self.date_to_be_formatted = template.Variable(date_to_be
_formatted)
        self.format_string = format_string

    def render(self, context):
        try:
            actual_date = self.date_to_be_formatted.resolve(cont
ext)
            return actual_date.strftime(self.format_string)
        except template.VariableDoesNotExist:
            return ''
```

如果变量分辨率无法解析在页面的当前上下文中传递给它的字符串，则会抛出 `VariableDoesNotExist` 异常。

在上下文中设置变量

上面的例子只是输出一个值。通常，如果您的模板标记设置模板变量而不是输出值，则更灵活。这样，模板作者可以重用您的模板标签创建的值。

要在上下文中设置变量，只需对 `render()` 方法中的上下文对象使用字典分配。以下是更新版本的 `CurrentTimeNode` ，它设置模板变量 `current_time` ，而不是输出：

```
import datetime
from django import template

class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        context['current_time'] = datetime.datetime.now().strfti
me(self.format_string)
        return ''
```

请注意， `render()` 返回空字符串。 `render()` 应始终返回字符串输出。如果所有模板标签都设置了一个变量， `render()` 应该返回空字符串。

以下说明如何使用这个新版本的标记：

```
{% current_time "%Y-%M-%d %I:%M %p" %}<p>The time is {{ current_
time }}.</p>
```

上下文中的变量范围

上下文中的任何变量集都只能在分配给它的模板的相同 `block` 中使用。这种行为是故意的；它为变量提供了一个范围，使它们不与其他块中的上下文冲突。

但是， `CurrentTimeNode2` 有一个问题：变量名称 `current_time` 是硬编码的。这意味着您需要确保您的模板在其他地方不使用 `{{ current_time }}` ，因为 `}} current_time }}` 会盲目地覆盖该变量的值。一个更清洁的解决方案是使模板标签指定输出变量的名称，如下所示：

```
{% current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

为此，您需要重构编译函数和 `Node` 类，如下所示：

```
import re

class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name
    def render(self, context):
        context[self.var_name] = datetime.datetime.now().strftim
e(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag conten
ts.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        raise template.TemplateSyntaxError(
            "%r tag requires arguments" % token.contents.split()
[0]
        )
    m = re.search(r'(.*?) as (\w+)', arg)
    if not m:
        raise template.TemplateSyntaxError("%r tag had invalid a
rguments" % tag_name)
    format_string, var_name = m.groups()
    if not (format_string[0] == format_string[-1] and format_str
ing[0] in ('"', "'")):
        raise template.TemplateSyntaxError(
            "%r tag's argument should be in quotes" % tag_name
        )
    return CurrentTimeNode3(format_string[1:-1], var_name)
```

这里的区别在于 `do_current_time()` 抓取格式字符串和变量名称，将两者传递到 `CurrentTimeNode3` 。

最后，如果您只需要为自定义上下文更新模板标记提供一个简单的语法，则可以考虑使用上面介绍的*assignment tag*快捷方式。

解析直到另一个块标记

模板标签可以协同工作。例如，标签 `}} comment }}` 标签隐藏所有内容，直到 `}} endcomment }}` 。要创建这样的模板标签，请在编译函数中使用 `parser.parse()` 。

以下是简化的 `}} 注释 }}` 标记的实现方法：

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

注意

`}} comment }}` 的实际执行方式略有不同， `}} 注释 }}` 和 `}} }}` 。它通过调用 `parser.skip_past('endcomment')` 而不是 `parser.parse(('endcomment',))` ，然后紧跟 `parser.delete_first_token()` ，从而避免生成节点列表。

`parser.parse()` 使用块标签的名称的元组来解析，直到“”。它返回一个 `django.template.NodeList` 的实例，它是所有 `Node` 对象的列表，解析器遇到“before”遇到任何名为的标签元组。

In `"nodelist = parser.parse(('endcomment',))"` in the above example, `nodelist` is a list of all nodes between the `}} comment }}` and `}} endcomment }}` , not counting `}} comment }}` and `}} endcomment }}` themselves.

调用 `parser.parse()` 后，解析器尚未“消耗” `}} endcomment } }` 标记，因此代码需要显式调用 `parser.delete_first_token()` 。

`CommentNode.render()` 只返回一个空字符串。 `}} 注释之间的任何内容 }}` 和 `}} endcomment }}` 被忽略。

解析直到另一个块标签，并保存内容

在上一个示例中， `do_comment()` 舍弃了 `}} 注释之间的所有内容 }} t2 >和` }} endcomment }} `。` 而不是这样做，可以用块标记之间的代码做一些事情。

For example, here’s a custom template tag, `}} upper }}` , that capitalizes everything between itself and `}} endupper }}` .

用法：

```
{% upper %}This will appear in uppercase, {{ your_name }}.{% end
upper %}
```

和前面的例子一样，我们使用 `parser.parse()` 。但是这次，我们将得到的 `节点` 传递给 `节点` ：

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist
    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

这里唯一的新概念是`UpperNode.render()`中的`self.nodelist.render（context）`。

For more examples of complex rendering, see the source code of `}} for }}` in `django/template/defaulttags.py` and `}} if }}` in `django/template/smartif.py`.

表单

Django 提供了一个丰富的框架可便利地创建表单及操作表单数据。

基础

使用表单

> 关于这页文档
>
> 这页文档简单介绍Web 表单的基本概念和它们在Django 中是如何处理的。关于表单API 某方面的细节，请参见表单 API、表单的字段和表单和字段的检验。

除非你计划构建的网站和应用只是发布内容而不接受访问者的输入，否则你将需要理解并使用表单。

Django 提供广泛的工具和库来帮助你构建表单来接收网站访问者的输入，然后处理以及响应输入。

HTML 表单

在HTML中，表单是位于 `<form>...</form>` 之间的元素的集合，它们允许访问者输入文本、选择选项、操作对象和控制等等，然后将信息发送回服务器。

某些表单的元素 —— 文本输入和复选框 —— 非常简单而且内建于HTML 本身。其它的表单会复杂些；例如弹出一个日期选择对话框的界面、允许你移动滚动条的界面、使用JavaScript 和CSS 以及HTML 表单 `<input>` 元素来实现操作控制的界面。

与 `<input>` 元素一样，一个表单必须指定两样东西：

- where：响应用户输入的URL
- how：HTTP 方法

例如，Django Admin 站点的登录表单包含几个 `<input>` 元素： `type="text"` 用于用户名， `type="password"` 用于密码， `type="submit"` 用于“Log in" 按钮。它还包含一些用户看不到的隐藏的文本字段，Django 使用它们来决定下一步的行为。

它还告诉浏览器表单数据应该发往 `<form>` 的 `action` 属性指定的URL —— `/admin/` ，而且应该使用 `method` 属性指定的HTTP 方法 —— `post` 。

当触发 `<input type="submit" value="Log in">` 元素时，数据将发送给 `/admin/` 。

GET 和 POST

处理表单时候只会用到 `GET` 和 `POST` 方法。

Django 的登录表单使用POST 方法，在这个方法中浏览器组合表单数据、对它们进行编码以用于传输、将它们发送到服务器然后接收它的响应。

相反，GET 组合提交的数据为一个字符串，然后使用它来生成一个URL。这个URL 将包含数据发送的地址以及数据的键和值。如果你在Django 文档中做一次搜索，你会立即看到这点，此时将生成一个 `https://docs.djangoproject.com/search/?q=forms&release=1` 形式的 URL。

`GET` 和 `POST` 用于不同的目的。

用于改变系统状态的请求 —— 例如，给数据库带来变化的请求 —— 应该使用 `POST`。`GET` 只应该用于不会影响系统状态的请求。

`GET` 还不适合密码表单，因为密码将出现在URL 中，以及浏览器的历史和服务器的日志中，而且都是以普通的文本格式。它还不适合数据量大的表单和二进制数据，例如一张图片。使用GET 请求作为管理站点的表单具有安全隐患：攻击者很容易模拟表单请求来取得系统的敏感数据。`POST`，如果与其它的保护措施结合将对访问提供更多的控制，例如Django 的CSRF 保护。

另一个方面，`GET` 适合网页搜索这样的表单，因为这种表示一个 `GET` 请求的 URL 可以很容易地作为书签、分享和重新提交。

Django 在表单中的角色

处理表单是一件很复杂的事情。考虑一下Django 的Admin 站点，不同类型的大量数据项需要在一个表单中准备好、渲染成HTML、使用一个方便的界面编辑、返回给服务器、验证并清除，然后保存或者向后继续处理。

Django 的表单功能可以简化并自动化大部分这些工作，而且还可以比大部分程序员自己所编写的代码更安全。

Django 会处理表单工作中的三个显著不同的部分：

- 准备并重新构造数据
- 为数据创建HTML 表单
- 接收并处理客户端提交的表单和数据

可以手工编写代码来实现，但是Django 可以帮你完成所有这些工作。

Django 中的表单

我们已经简短讲述HTML 表单，但是HTML的 `<form>` 只是其机制的一部分。

在一个Web 应用中，'表单'可能指HTML `<form>`、或者生成它的Django 的 `Form`、或者提交时发送的结构化数据、或者这些部分的总和。

Django 的Form 类

表单系统的核心部分是Django 的 `Form` 类。Django 的模型描述一个对象的逻辑结构、行为以及展现给我们的方式，与此类似，`Form` 类描述一个表单并决定它如何工作和展现。

模型类的字典映射到数据库的字典，与此类似，表单类的字段映射到HTML 的表单 `<input>` 元素。（ `ModelForm` 通过一个 `Form` 映射模型类的字段到HTML 表单的 `<input>` 元素；Django 的Admin 站点就是基于这个）。

表单的字段本身也是类；它们管理表单的数据并在表单提交时进行验证。 `DateField` 和 `FileField` 处理的数据类型差别很大，必须完成不同的事情。

表单字段在浏览器中呈现给用户的是一个HTML 的“widget” —— 用户界面的一个片段。每个字段类型都有一个合适的默认Widget 类，需要时可以覆盖。

实例化、处理和渲染表单

在Django 中渲染一个对象时，我们通常：

1. 在视图中获得它（例如，从数据库中获取）
2. 将它传递给模板上下文
3. 使用模板变量将它扩展为HTML 标记

在模板中渲染表单和渲染其它类型的对象几乎一样，除了几个关键的差别。

在模型实例不包含数据的情况下，在模板中对它做处理很少有什么用处。但是渲染一个未填充的表单却非常有意义 —— 我们希望用户去填充它。

所以当我们在视图中处理模型实例时，我们一般从数据库中获取它。当我们处理表单时，我们一般在视图中实例化它。

当我们实例化表单时，我们可以选择让它为空还是预先填充它，例如使用：

- 来自一个保存后的模型实例的数据（例如用于编辑的管理表单）
- 我们从其它地方获得的数据
- 从前面一个HTML 表单提交过来的数据

最后一种情况最令人关注，因为它使得用户可以不只是阅读一个网站，而且可以给网站返回信息。

构建一个表单

需要完成的工作

假设你想在你的网站上创建一个简单的表单，以获得用户的名字。你需要类似这样的模板：

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{
 current_name }}">
    <input type="submit" value="OK">
</form>
```

这告诉浏览器发送表单的数据到URL `/your-name/` ，并使用 `POST` 方法。它将显示一个标签为"Your name:"的文本字段，和一个"OK"按钮。如果模板上下文包含一个 `current_name` 变量，它将用于预填充 `your_name` 字段。

你将需要一个视图来渲染这个包含HTML 表单的模板，并提供合适的 `current_name` 字段。

当表单提交时，发往服务器的 `POST` 请求将包含表单数据。

现在你还需要一个对应 `/your-name/` URL 的视图，它在请求中找到正确的键/值对，然后处理它们。

这是一个非常简单的表单。实际应用中，一个表单可能包含几十上百个字段，其中大部分需要预填充，而且我们预料到用户将来回编辑-提交几次才能完成操作。

我们可能需要在表单提交之前，在浏览器端作一些验证。我们可能想使用非常复杂的字段，以允许用户做类似从日历中挑选日期这样的事情，等等。

这个时候，让Django 来为我们完成大部分工作是很容易的。

在Django 中构建一个表单

Form 类

我们已经计划好了我们的 HTML 表单应该呈现的样子。在Django 中，我们的起始点是这里：

```
#forms.py

from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=10
0)
```

它定义一个 `Form` 类，只带有一个字段（ `your_name` ）。我们已经对这个字段使用一个友好的标签，当渲染时它将出现在 `<label>` 中（在这个例子中，即使我们省略它，我们指定的 `label` 还是会自动生成）。

字段允许的最大长度通过 `max_length` 定义。它完成两件事情。首先，它在HTML的 `<input>` 上放置一个 `maxlength="100"` （这样浏览器将在第一时间阻止用户输入多于这个数目的字符）。它还意味着当Django 收到浏览器发送过来的表单时，它将验证数据的长度。

`Form` 的实例具有一个 `is_valid()` 方法，它为所有的字段运行验证的程序。当调用这个方法时，如果所有的字段都包含合法的数据，它将：

- 返回 `True`
- 将表单的数据放到 `cleaned_data` 属性中。

完整的表单，第一次渲染时，看上去将像：

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="10
0">
```

注意它不包含 `<form>` 标签和提交按钮。我们必须自己在模板中提供它们。

视图

发送给Django 网站的表单数据通过一个视图处理，一般和发布这个表单的是同一个视图。这允许我们重用一些相同的逻辑。

当处理表单时，我们需要在视图中实例化它：

```
#views.py

from django.shortcuts import render
from django.http import HttpResponseRedirect

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from
the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect('/thanks/')

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})
```

如果访问视图的是一个 `GET` 请求，它将创建一个空的表单实例并将它放置到要渲染的模板的上下文中。这是我们在第一个访问该URL 时预期发生的情况。

如果表单的提交使用 `POST` 请求，那么视图将再次创建一个表单实例并使用请求中的数据填充它： `form = NameForm(request.POST)` 。这叫做"绑定数据至表单“（它现在是一个绑定的表单）。

我们调用表单的 `is_valid()` 方法；如果它不为 `True` ，我们将带着这个表单返回到模板。这时表单不再为空（未绑定），所以HTML 表单将用之前提交的数据填充，然后可以根据要求编辑并改正它。

如果 `is_valid()` 为 `True` ，我们将能够在 `cleaned_data` 属性中找到所有合法的表单数据。在发送HTTP 重定向给浏览器告诉它下一步的去向之前，我们可以用这个数据来更新数据库或者做其它处理。

模板

我们不需要在name.html 模板中做很多工作。最简单的例子是：

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

根据 `{{ form }}` ，所有的表单字段和它们的属性将通过Django 的模板语言拆分成HTML 标记 。

> 表单和跨站请求伪造的防护
>
> Django 原生支持一个简单易用的跨站请求伪造的防护。当提交一个启用CSRF 防护的 `POST` 表单时，你必须使用上面例子中的 `csrf_token` 模板标签。然而，因为CSRF 防护在模板中不是与表单直接捆绑在一起的，这个标签在这篇文档的以下示例中将省略。
>
> HTML5 输入类型和浏览器验证
>
> 如果你的表单包含 `URLField` 、 `EmailField` 和其它整数字段类似，Django 将使用 `url` 、 `email` 和 `number` 这样的HTML5 输入类型。默认情况下，浏览器可能会对这些字段进行它们自身的验证，这些验证可能比Django 的验证更严格。如果你想禁用这个行为，请设置 `form` 标签的 `novalidate` 属性，或者指定一个不同的字段，如 `TextInput` 。

现在我们有了一个可以工作的网页表单，它通过Django Form 描述、通过视图处理并渲染成一个HTML `<form>` 。

这是你入门所需要知道的所有内容，但是表单框架为了提供了更多的内容。一旦你理解了上面描述的基本处理过程，你应该可以理解表单系统的其它功能并准备好学习更多的底层机制。

Django Form 类详解

所有的表单类都作为 `django.forms.Form` 的子类创建，包括你在Django 管理站点中遇到的 `ModelForm` 。

> 模型和表单
>
> 实际上，如果你的表单打算直接用来添加和编辑Django 的模型，ModelForm 可以节省你的许多时间、精力和代码，因为它将根据 `Model` 类构建一个表单以及适当的字段和属性。

绑定的和未绑定的表单实例

绑定的和未绑定的表单 之间的区别非常重要：

- 未绑定的表单没有关联的数据。当渲染给用户时，它将为空或包含默认的值。
- 绑定的表单具有提交的数据，因此可以用来检验数据是否合法。如果渲染一个

不合法的绑定的表单，它将包含内联的错误信息，告诉用户如何纠正数据。

表单的 `is_bound` 属性将告诉你一个表单是否具有绑定的数据。

字段详解

考虑一个比上面的迷你示例更有用的一个表单，我们可以用它来在一个个人网站上实现“联系我”功能：

```
#forms.py

from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

我们前面的表单只使用一个字段 `your_name` ，它是一个 `CharField` 。在这个例子中，我们的表单具有四个字段： `subject` 、 `message` 、 `sender` 和 `cc_myself` 。共用到三种字段类型： `CharField` 、 `EmailField` 和 `BooleanField` ；完整的字段类型列表可以在表单字段中找到。

Widgets

每个表单字段都有一个对应的 `Widget` 类，它对应一个HTML 表单 `Widget` ，例如 `<input type="text">` 。

在大部分情况下，字段都具有一个合理的默认Widget。例如，默认情况下， `CharField` 具有一个 `TextInput Widget` ，它在HTML 中生成一个 `<input type="text">` 。如果你需要 `<textarea>` ，在定义表单字段时你应该指定一个合适的 `Widget` ，例如我们定义的 `message` 字段。

字段的数据

不管表单提交的是什么数据，一旦通过调用 `is_valid()` 成功验证（ `is_valid()` 返回 `True` ），验证后的表单数据将位于 `form.cleaned_data` 字典中。这些数据已经为你转换好为Python 的类型。

> 注
>
> 此时，你依然可以从 `request.POST` 中直接访问到未验证的数据，但是访问验证后的数据更好一些。

在上面的联系表单示例中， `cc_myself` 将是一个布尔值。类似地， `IntegerField` 和 `FloatField` 字段分别将值转换为Python 的 `int` 和 `float` 。

下面是在视图中如何处理表单数据：

```
#views.py

from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

> 提示
>
> 关于Django 中如何发送邮件的更多信息，请参见发送邮件。

有些字段类型需要一些额外的处理。例如，使用表单上传的文件需要不同地处理（它们可以从 `request.FILES` 获取，而不是 `request.POST` ）。如何使用表单处理文件上传的更多细节，请参见绑定上传的文件到一个表单。

使用表单模板

你需要做的就是将表单实例放进模板的上下文。如果你的表单在 `Contex` t 中叫做 `form` ，那么 `{{ form }}` 将正确地渲染它的 `<label>` 和 `<input>` 元素。

表单渲染的选项

> 表单模板的额外标签
>
> 不要忘记，表单的输出不 包含 `<form>` 标签，和表单的 `submit` 按钮。你必须自己提供它们。

对于 `<label>/<input>` 对，还有几个输出选项：

- `{{ form.as_table }}` 以表格的形式将它们渲染在 `<tr>` 标签中
- `{{ form.as_p }}` 将它们渲染在 `<p>` 标签中
- `{{ form.as_ul }}` 将它们渲染在 `<li>` 标签中

注意，你必须自己提供 `<table>` 或 `<ul>` 元素。

下面是我们的 `ContactForm` 实例的输出 `{{ form.as_p }}` ：

```
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength=
"100" /></p>
<p><label for="id_message">Message:</label>
    <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label>
    <input type="email" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself" />
</p>
```

注意，每个表单字段具有一个 `ID` 属性并设置为 `id_<field-name>` ，它被一起的 `label` 标签引用。它对于确保屏幕阅读软件这类的辅助计算非常重要。你还可以自定义label 和 id 生成的方式。

更多信息参见 输出表单为HTML。

手工渲染字段

我们没有必要非要让Django 来分析表单的字段；如果我们喜欢，我们可以手工来做（例如，这样允许重新对字段排序）。每个字段都是表单的一个属性，可以使用 `{{ form.name_of_field }}` 访问，并将在Django 模板中正确地渲染。例如：

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:<
/label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</
label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email addre
ss:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?<
/label>
    {{ form.cc_myself }}
</div>
```

完整的 `<label>` 元素还可以使用 `label_tag()` 生成。例如：

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

渲染表单的错误信息

当然，这个便利性的代价是更多的工作。直到现在，我们没有担心如何展示错误信息，因为Django 已经帮我们处理好。在下面的例子中，我们将自己处理每个字段的错误和表单整体的各种错误。注意，表单和模板顶部的 `{{ form.non_field_errors }}` 查找每个字段的错误。

使用 `{{ form.name_of_field.errors }}` 显示表单错误的一个清单，并渲染成一个 `ul` 。看上去可能像：

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

这个 `ul` 有一个 `errorlist` CSS 类型，你可以用它来定义外观。如果你希望进一步自定义错误信息的显示，你可以迭代它们来实现：

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

空字段错误（以及使用 `form.as_p()` 时渲染的隐藏字段错误）将渲染成一个额外的CSS 类型 `nonfield` 以帮助区分每个字段的错误信息。例如，`{{ form.non_field_errors }}` 看上去会像：

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

```
Changed in Django 1.8:

添加上面示例中提到的nonfield CSS 类型。
```

参见Forms API 以获得关于错误、样式以及在模板中使用表单属性的更多内容。

迭代表单的字段

如果你为你的表单使用相同的HTML，你可以使用 `{% for %}` 循环迭代每个字段来减少重复的代码：

```
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

`{{ field }}` 中有用的属性包括：

`{{ field.label }}`

字段的 `label` ，例如 `Email address` 。

`{{ field.label_tag }}`

包含在HTML `<label>` 标签中的字段 `Label` 。它包含表单的 `label_suffix` 。例如，默认的 `label_suffix` 是一个冒号：

```
<label for="id_email">Email address:</label>
```

`{{ field.id_for_label }}`

用于这个字段的 `ID` （在上面的例子中是 `id_email` ）。如果你正在手工构造 `label` ，你可能想使用它代替 `label_tag` 。如果你有一些内嵌的JavaScript并且想避免硬编码字段的 `ID` ，这也是有用的。

`{{ field.value }}`

字段的值，例如 `someone@example.com` 。

`{{ field.html_name }}`

输入元素的 `name` 属性中将使用的名称。它将考虑到表单的前缀。

`{{ field.help_text }}`

与该字段关联的帮助文档。

`{{ field.errors }}`

输出一个 `<ul class="errorlist">` ，包含这个字段的验证错误信息。你可以使用 `{% for error in field.errors %}` 自定义错误的显示。 这种情况下，循环中的每个对象只是一个包含错误信息的简单字符串。

`{{ field.is_hidden }}`

如果字段是隐藏字段，则为 `True` ，否则为 `False` 。作为模板变量，它不是很有用处，但是可以用于条件测试，例如：

```
{% if field.is_hidden %}

{% endif %}
```

`{{ field.field }}`

表单类中的 `Field` 实例，通过 `BoundField` 封装。你可以使用它来访问 `Field` 属性，例如 `{% char_field.field.max_length %}`。

迭代隐藏和可见的字段

如果你正在手工布局模板中的一个表单，而不是依赖Django 默认的表单布局，你可能希望将 `<input type="hidden">` 字段与非隐藏的字段区别对待。例如，因为隐藏的字段不会显示，在该字段旁边放置错误信息可能让你的用户感到困惑——所以这些字段的错误应该有区别地来处理。

Django 提供两个表单方法，它们允许你独立地在隐藏的和可见的字段上迭代：`hidden_fields()` 和 `visible_fields()`。下面是使用这两个方法对前面一个例子的修改：

```
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}

{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

这个示例没有处理隐藏字段中的任何错误信息。通常，隐藏字段中的错误意味着表单被篡改，因为正常的表单填写不会改变它们。然而，你也可以很容易地为这些表单错误插入一些错误信息显示出来。

可重用的表单模板

如果你的网站在多个地方对表单使用相同的渲染逻辑，你可以保存表单的循环到一个单独的模板中来减少重复，然后在其它模板中使用 `include` 标签来重用它：

```
# In your form template:
{% include "form_snippet.html" %}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

如果传递到模板上下文中的表单对象具有一个不同的名称，你可以使用 `include` 标签的 `with` 参数来对它起个别名：

```
{% include "form_snippet.html" with form=comment_form %}
```

如果你发现自己经常这样做，你可能需要考虑一下创建一个自定义的 `inclusion` 标签。

更深入的主题

这里只是基础，表单还可以完成更多的工作：

- 表单集
  - 在表单集中使用初始化数据
  - 限制表单的最大数目
  - 表单集的验证
  - 验证表单集中表单的数目
  - 处理表单的排序和删除
  - 添加额外的字段到表单中
  - 在视图和模板中视图表单集
- 从模型中创建表单
  - ModelForm
  - 模型表单集
  - Inline formsets
- 表单集（`Media` 类）
  - Assets as a static definition
  - `Media` as a dynamic property
  - Paths in asset definitions
  - `Media` 对象
  - 表单中的 `Media`

> 另见
>
> 表单参考 覆盖完整的API 参考，包括表单字段、表单Widget 以及表单和字段的验证。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

表单 API

关于这篇文档

这篇文档讲述Django 表单API 的详细细节。你应该先阅读表单简介。

绑定的表单和未绑定的表单

`表单` 要么是绑定的，要么是未绑定的。

- 如果是绑定的，那么它能够验证数据，并渲染表单及其数据成HTML。
- 如果是未绑定的，那么它不能够完成验证（因为没有可验证的数据！），但是仍然能渲染空白的表单成HTML。

class `Form`

若要创建一个未绑定的 `表单` 实例，只需简单地实例化该类：

```
>>> f = ContactForm()
```

若要绑定数据到表单，可以将数据以字典的形式传递给 `表单` 类的构造函数的第一个参数：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
```

在这个字典中，键为字段的名称，它们对应于 `表单` 类中的属性。值为需要验证的数据。它们通常为字符串，但是没有强制要求必须是字符串；传递的数据类型取决于 `字段` ，我们稍后会看到。

`Form.``is_bound`

如果运行时刻你需要区分绑定的表单和未绑定的表单，可以检查下表单 `is_bound` 属性的值：

```
>>> f = ContactForm()
>>> f.is_bound
False
>>> f = ContactForm({'subject': 'hello'})
>>> f.is_bound
True
```

注意，传递一个空的字典将创建一个带有空数据的绑定的表单：

```
>>> f = ContactForm({})
>>> f.is_bound
True
```

如果你有一个绑定的 `表单` 实例但是想改下数据，或者你想绑定一个未绑定的 `表单` 表单到某些数据，你需要创建另外一个 `表单` 实例。 `Form` 实例的数据没有办法修改。 `表单` 实例一旦创建，你应该将它的数据视为不可变的，无论它有没有数据。

使用表单来验证数据

`Form.``clean` ()

当你需要为相互依赖的字段添加自定义的验证时，你可以实现 `表单` 的 `clean()` 方法。示例用法参见*Cleaning and validating fields that depend on each other*。

`Form.``is_valid` ()

`表单` 对象的首要任务就是验证数据。对于绑定的 `表单` 实例，可以调用 `is_valid()` 方法来执行验证并返回一个表示数据是否合法的布尔值。

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
```

让我们试下非法的数据。下面的情形中， `subject` 为空（默认所有字段都是必需的）且 `sender` 是一个不合法的邮件地址：

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
```

`Form.``errors`

访问 `errors` 属性可以获得错误信息的一个字典：

```
>>> f.errors
{'sender': ['Enter a valid email address.'], 'subject': ['This field is required.']}
```

在这个字典中，键为字段的名称，值为表示错误信息的Unicode 字符串组成的列表。错误信息保存在列表中是因为字段可能有多个错误信息。

你可以在调用 `is_valid()` 之前访问 `errors` 。表单的数据将在第一次调用 `is_valid()` 或者访问 `errors` 时验证。

验证将值调用一次，无论你访问 `errors` 或者调用 `is_valid()` 多少次。这意味着，如果验证过程有副作用，这些副作用将只触发一次。

`Form.errors.``as_data` ()

New in Django 1.7.

返回一个 `字典` ，它映射字段到原始的 `ValidationError` 实例。

```
>>> f.errors.as_data()
{'sender': [ValidationError(['Enter a valid email address.'])],
'subject': [ValidationError(['This field is required.'])]}
```

每当你需要根据错误的 `code` 来识别错误时，可以调用这个方法。它可以用来重写错误信息或者根据特定的错误编写自定义的逻辑。它还可以用来序列化错误为一个自定义的格式（例如，XML）； `as_json()` 就依赖于 `as_data()` 。

需要 `as_data()` 方法是为了向后兼容。以前， `ValidationError` 实例在它们渲染后 的错误消息一旦添加到 `Form.errors` 字典就立即被丢弃。理想情况下， `Form.errors` 应该已经保存 `ValidationError` 实例而带有 `as_` 前缀的方法可以渲染它们，但是为了不破坏直接使用 `Form.errors` 中的错误消息的代码，必须使用其它方法来实现。

`Form.errors.``as_json` (*escape_html=False*)

New in Django 1.7.

返回JSON 序列化后的错误。

```
>>> f.errors.as_json()
{"sender": [{"message": "Enter a valid email address.", "code": "invalid"}],
"subject": [{"message": "This field is required.", "code": "required"}]}
```

默认情况下， `as_json()` 不会转义它的输出。如果你正在使用AJAX 请求表单视图，而客户端会解析响应并将错误插入到页面中，你必须在客户端对结果进行转义以避免可能的跨站脚本攻击。使用一个JavaScript 库比如jQuery 来做这件事很简单 —— 只要使用 `$(el).text(errorText)` 而不是 `.html()` 就可以。

如果由于某种原因你不想使用客户端的转义，你还可以设置 `escape_html=True` ，这样错误消息将被转义而你可以直接在HTML 中使用它们。

`Form.``add_error` (*field*, *error*)

New in Django 1.7.

这个方法允许在 `Form.clean()` 方法内部或从表单的外部一起给字段添加错误信息；例如从一个视图中。

`field` 参数为字段的名称。如果值为 `None` ，error 将作为 `Form.non_field_errors()` 返回的一个非字段错误。

`error` 参数可以是一个简单的字符串，或者最好是一个 `ValidationError` 实例。引发*ValidationError* 中可以看到定义表单错误时的最佳实践。

注意， `Form.add_error()` 会自动删除 `cleaned_data` 中的相关字段。

`Form.``has_error` (*field*, *code=None*)

New in Django 1.8.

这个方法返回一个布尔值，指示一个字段是否具有指定错误 `code` 的错误。当 `code` 为 `None` 时，如果字段有任何错误它都将返回 `True` 。

若要检查非字段错误，使用 `NON_FIELD_ERRORS` 作为 `field` 参数。

`Form.``non_field_errors` ()

这个方法返回 `Form.errors` 中不是与特定字段相关联的错误。它包含在 `Form.clean()` 中引发的 `ValidationError` 和使用 `Form.add_error(None, "...")` 添加的错误。

未绑定表单的行为

验证没有绑定数据的表单是没有意义的，下面的例子展示了这种情况：

```
>>> f = ContactForm()
>>> f.is_valid()
False
>>> f.errors
{}
```

动态的初始值

`Form.``initial`

表单字段的初始值使用 `initial` 声明。例如，你可能希望使用当前会话的用户名填充 `username` 字段。

使用 `Form` 的 `initial` 参数可以实现。该参数是字段名到初始值的一个字典。只需要包含你期望给出初始值的字段；不需要包含表单中的所有字段。例如：

```
>>> f = ContactForm(initial={'subject': 'Hi there!'})
```

这些值只显示在没有绑定的表单中，即使没有提供特定值它们也不会作为后备的值。

注意，如果 字段 有定义 `initial` ，而实例化 表单 时也提供 `initial` ，那么后面的 `initial` 将优先。在下面的例子中， `initial` 在字段和表单实例化中都有定义，此时后者具有优先权：

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='class')
...     url = forms.URLField()
...     comment = forms.CharField()
>>> f = CommentForm(initial={'name': 'instance'}, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="inst
ance" /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></t
d></tr>
```

检查表单数据是否改变

`Form.``has_changed` ()

当你需要检查表单的数据是否从初始数据发生改变时，可以使用 表单 的 `has_changed()` 方法。

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data, initial=data)
>>> f.has_changed()
False
```

当提交表单时，我们可以重新构建表单并提供初始值，这样可以实现比较：

```
>>> f = ContactForm(request.POST, initial=data)
>>> f.has_changed()
```

如果 `request.POST` 中的数据与 `initial` 中的不同， `has_changed()` 将为 `True` ，否则为 `False` 。计算的结果是通过调用表单每个字段的 `Field.has_changed()` 得到的。

从表单中访问字段

`Form.``fields`

你可以从 表单 实例的 `fields` 属性访问字段：

```
>>> for row in f.fields.values(): print(row)
...
<django.forms.fields.CharField object at 0x7ffaac632510>
<django.forms.fields.URLField object at 0x7ffaac632f90>
<django.forms.fields.CharField object at 0x7ffaac3aa050>
>>> f.fields['name']
<django.forms.fields.CharField object at 0x7ffaac6324d0>
```

可你可以修改 表单 实例的字段来改变字段在表单中的表示：

```
>>> f.as_table().split('\n')[0]
'<tr><th>Name:</th><td><input name="name" type="text" value="ins
tance" /></td></tr>'
>>> f.fields['name'].label = "Username"
>>> f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value=
"instance" /></td></tr>'
```

注意不要改变 `base_fields` 属性，因为一旦修改将影响同一个Python 进程中接下来所有的 `ContactForm` 实例：

```
>>> f.base_fields['name'].label = "Username"
>>> another_f = CommentForm(auto_id=False)
>>> another_f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value=
"class" /></td></tr>'
```

访问“清洁”的数据

`Form.``cleaned_data`

表单 类中的每个字段不仅负责验证数据，还负责“清洁”它们 —— 将它们转换为正确的格式。这是个非常好用的功能，因为它允许字段以多种方式输入数据，并总能得到一致的输出。

例如，`DateField` 将输入转换为Python 的 `datetime.date` 对象。无论你传递的是 `'1994-07-15'` 格式的字符串、`datetime.date` 对象、还是其它格式的数字，`DateField` 将始终将它们转换成 `datetime.date` 对象，只要它们是合法的。

一旦你创建一个 表单 实例并通过验证后，你就可以通过它的 `cleaned_data` 属性访问清洁的数据：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'cc_myself': True, 'message': 'Hi there', 'sender': 'foo@exampl
e.com', 'subject': 'hello'}
```

注意，文本字段 —— 例如，`CharField` 和 `EmailField` —— 始终将输入转换为Unicode 字符串。我们将在这篇文档的后面将是编码的影响。

如果你的数据没有 通过验证，`cleaned_data` 字典中只包含合法的字段：

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
>>> f.cleaned_data
{'cc_myself': True, 'message': 'Hi there'}
```

`cleaned_data` 始终只 包含 `表单` 中定义的字段，即使你在构建 `表单` 时传递了额外的数据。在下面的例子中，我们传递一组额外的字段给 `ContactForm` 构造函数，但是 `cleaned_data` 将只包含表单的字段：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True,
...         'extra_field_1': 'foo',
...         'extra_field_2': 'bar',
...         'extra_field_3': 'baz'}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data # Doesn't contain extra_field_1, etc.
{'cc_myself': True, 'message': 'Hi there', 'sender': 'foo@exampl
e.com', 'subject': 'hello'}
```

当 `表单` 合法时， `cleaned_data` 将包含所有字段的键和值，即使传递的数据不包含某些可选字段的值。在下面的例子中，传递的数据字典不包含 `nick_name` 字段的值，但是 `cleaned_data` 任然包含它，只是值为空：

```
>>> from django.forms import Form
>>> class OptionalPersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
...     nick_name = CharField(required=False)
>>> data = {'first_name': 'John', 'last_name': 'Lennon'}
>>> f = OptionalPersonForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'nick_name': '', 'first_name': 'John', 'last_name': 'Lennon'}
```

在上面的例子中， `cleaned_data` 中 `nick_name` 设置为一个空字符串，这是因为 `nick_name` 是 `CharField` 而 `CharField` 将空值作为一个空字符串。每个字段都知道自己的“空”值 —— 例如， `DateField` 的空值是 `None` 而不是一个空字符串。关于每个字段空值的完整细节，参见“内建的 `Field` 类”一节中每个字段的“空值”提示。

你可以自己编写代码来对特定的字段（根据它们的名字）或者表单整体（考虑到不同字段的组合）进行验证。更多信息参见表单和字段验证。

输出表单为HTML

`表单` 对象的第二个任务是将它渲染成HTML。很简单， `print` 它：

```
>>> f = ContactForm()
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input
id="id_subject" type="text" name="subject" maxlength="100" /></t
d></tr>
<tr><th><label for="id_message">Message:</label></th><td><input
type="text" name="message" id="id_message" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input ty
pe="email" name="sender" id="id_sender" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><in
put type="checkbox" name="cc_myself" id="id_cc_myself" /></td></
tr>
```

如果表单是绑定的，输出的HTML 将包含数据。例如，如果字段是 `<input type="text">` 的形式，其数据将位于 `value` 属性中。如果字段是 `<input type="checkbox">` 的形式，HTML 将包含 `checked="checked"` ：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input
id="id_subject" type="text" name="subject" maxlength="100" value
="hello" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input
type="text" name="message" id="id_message" value="Hi there" /></
td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input ty
pe="email" name="sender" id="id_sender" value="foo@example.com"
/></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><in
put type="checkbox" name="cc_myself" id="id_cc_myself" checked="
checked" /></td></tr>
```

默认的输出时具有两个列的HTML 表格，每个字段对应一个 `<tr>` 。注意事项：

- 为了灵活性，输出不包含 `<table>` 和 `</table>` 、 `<form>` 和 `</form>` 以及 `<input type="submit">` 标签。你需要添加它们。
- 每个字段类型有一个默认的HTML 表示。 `CharField` 表示为一个 `<input type="text">` ， `EmailField` 表示为一个 `<input type="email">` 。 `BooleanField` 表示为一个 `<input type="checkbox">` 。注意，这些只是默认的表示；你可以使用Widget 指定字段使用哪种HTML，我们将稍后解释。
- 每个标签的HTML `name` 直接从 `ContactForm` 类中获取。
- 每个字段的文本标签 —— 例如 `'Subject:'` 、 `'Message:'` 和 `'Cc myself:'` 通过将所有的下划线转换成空格并大写第一个字母生成。

再次提醒，这些只是默认的表示；你可以手工指定标签。
- 每个文本标签周围有一个HTML `<label>` 标签，它指向表单字段的 `id` 。这个 `id` ，是通过在字段名称前面加上 `'id_'` 前缀生成。 `id` 属性和 `<label>` 标签默认包含在输出中，但你可以改变这一行为。

虽然 `print` 表单时 `<table>` 是默认的输出格式，但是还有其它格式可用。每个格式对应于表单对象的一个方法，每个方法都返回一个Unicode 对象。

as_p()

`Form.``as_p` ()

`as_p()` 渲染表单为一系列的 `<p>` 标签，每个 `<p>` 标签包含一个字段：

```
>>> f = ContactForm()
>>> f.as_p()
'<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxlength="100" /></p>\n<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></p>\n<p><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></p>\n<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>'
>>> print(f.as_p())
<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```

as_ul()

`Form.``as_ul` ()

`as_ul()` 渲染表单为一系列的 `<li>` 标签，每个 `<li>` 标签包含一个字段。它不包含 `<ul>` 和 `</ul>` ，所以你可以自己指定 `<ul>` 的任何HTML 属性：

```
>>> f = ContactForm()
>>> f.as_ul()
'<li><label for="id_subject">Subject:</label> <input id="id_subj
ect" type="text" name="subject" maxlength="100" /></li>\n<li><la
bel for="id_message">Message:</label> <input type="text" name="m
essage" id="id_message" /></li>\n<li><label for="id_sender">Send
er:</label> <input type="email" name="sender" id="id_sender" /><
/li>\n<li><label for="id_cc_myself">Cc myself:</label> <input ty
pe="checkbox" name="cc_myself" id="id_cc_myself" /></li>'
>>> print(f.as_ul())
<li><label for="id_subject">Subject:</label> <input id="id_subje
ct" type="text" name="subject" maxlength="100" /></li>
<li><label for="id_message">Message:</label> <input type="text"
name="message" id="id_message" /></li>
<li><label for="id_sender">Sender:</label> <input type="email" n
ame="sender" id="id_sender" /></li>
<li><label for="id_cc_myself">Cc myself:</label> <input type="ch
eckbox" name="cc_myself" id="id_cc_myself" /></li>
```

as_table()

`Form.``as_table` ()

最后，`as_table()` 输出表单为一个HTML `<table>` 。它与 `print` 完全相同。事实上，当你 `print` 一个表单对象时，在后台调用的就是 `as_table()` 方法：

```
>>> f = ContactForm()
>>> f.as_table()
'<tr><th><label for="id_subject">Subject:</label></th><td><input
 id="id_subject" type="text" name="subject" maxlength="100" /></
td></tr>\n<tr><th><label for="id_message">Message:</label></th><
td><input type="text" name="message" id="id_message" /></td></tr
>\n<tr><th><label for="id_sender">Sender:</label></th><td><input
 type="email" name="sender" id="id_sender" /></td></tr>\n<tr><th
><label for="id_cc_myself">Cc myself:</label></th><td><input typ
e="checkbox" name="cc_myself" id="id_cc_myself" /></td></tr>'
>>> print(f.as_table())
<tr><th><label for="id_subject">Subject:</label></th><td><input
id="id_subject" type="text" name="subject" maxlength="100" /></t
d></tr>
<tr><th><label for="id_message">Message:</label></th><td><input
type="text" name="message" id="id_message" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input ty
pe="email" name="sender" id="id_sender" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><in
put type="checkbox" name="cc_myself" id="id_cc_myself" /></td></
tr>
```

表单必填行和错误行的样式

`Form.``error_css_class`

`Form.``required_css_class`

将必填的表单行和有错误的表单行定义不同的样式特别常见。例如，你想将必填的表单行以粗体显示、将错误以红色显示。

表单 类具有一对钩子，可以使用它们来添加 `class` 属性给必填的行或有错误的行：只需简单地设置 `Form.error_css_class` 和/或 `Form.required_css_class` 属性：

```
from django.forms import Form

class ContactForm(Form):
    error_css_class = 'error'
    required_css_class = 'required'

    # ... and the rest of your fields here
```

一旦你设置好，将根据需要设置行的 `"error"` 和/或 `"required"` CSS 类型。其HTML 看上去将类似：

```
>>> f = ContactForm(data)
>>> print(f.as_table())
<tr class="required"><th><label class="required" for="id_subject
">Subject:</label>    ...
<tr class="required"><th><label class="required" for="id_message
">Message:</label>    ...
<tr class="required error"><th><label class="required" for="id_s
ender">Sender:</label>      ...
<tr><th><label for="id_cc_myself">Cc myself:<label> ...
>>> f['subject'].label_tag()
<label class="required" for="id_subject">Subject:</label>
>>> f['subject'].label_tag(attrs={'class': 'foo'})
<label for="id_subject" class="foo required">Subject:</label>
```

Changed in Django 1.8:

`required_css_class` 添加到 `<label>` 标签，如上面所看到的。

配置表单元素的HTML `id` 属性和 `<label>` 标签

`Form.``auto_id`

默认情况下，表单的渲染方法包含：

- 表单元素的HTML `id` 属性
- 对应的 `<label>` 标签。HTML `<label>` 标签指示标签文本关联的表单元素。这个小小的改进让表单在辅助设备上具有更高的可用性。使用`<label>` 标签始终是个好想法。

`id` 属性值通过在表单字段名称的前面加上`id_` 生成。但是如果你想改变`id` 的生成方式或者完全删除 HTML `id` 属性和 `<label>` 标签，这个行为是可配置的。

`id` 和label 的行为使用`表单` 构造函数的`auto_id` 参数控制。这个参数必须为 `True` 、 `False` 或者一个字符串。

如果`auto_id` 为 `False` ，那么表单的输出将不包含`<label>` 标签和`id` 属性：

```
>>> f = ContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" /></td></tr>
<tr><th>Sender:</th><td><input type="email" name="sender" /></td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="email" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="email" name="sender" /></p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

如果`auto_id` 设置为`True` ，那么输出的表示将 包含`<label>` 标签并简单地使用字典名称作为每个表单字段的`id` ：

```
>>> f = ContactForm(auto_id=True)
>>> print(f.as_table())
<tr><th><label for="subject">Subject:</label></th><td><input id="subject" type="text" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="message">Message:</label></th><td><input type="text" name="message" id="message" /></td></tr>
<tr><th><label for="sender">Sender:</label></th><td><input type="email" name="sender" id="sender" /></td></tr>
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="100" /></li>
<li><label for="message">Message:</label> <input type="text" name="message" id="message" /></li>
<li><label for="sender">Sender:</label> <input type="email" name="sender" id="sender" /></li>
<li><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="cc_myself" /></li>
>>> print(f.as_p())
<p><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="message">Message:</label> <input type="text" name="message" id="message" /></p>
<p><label for="sender">Sender:</label> <input type="email" name="sender" id="sender" /></p>
<p><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="cc_myself" /></p>
```

如果 `auto_id` 设置为包含格式字符 `'%s'` 的字符串，那么表单的输出将包含 `<label>` 标签，并将根据格式字符串生成 `id` 属性。例如，对于格式字符串 `'field_%s'` ，名为 `subject` 的字段的 `id` 值将是 `'field_subject'` 。继续我们的例子：

```
>>> f = ContactForm(auto_id='id_for_%s')
>>> print(f.as_table())
<tr><th><label for="id_for_subject">Subject:</label></th><td><in
put id="id_for_subject" type="text" name="subject" maxlength="10
0" /></td></tr>
<tr><th><label for="id_for_message">Message:</label></th><td><in
put type="text" name="message" id="id_for_message" /></td></tr>
<tr><th><label for="id_for_sender">Sender:</label></th><td><inpu
t type="email" name="sender" id="id_for_sender" /></td></tr>
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td
><input type="checkbox" name="cc_myself" id="id_for_cc_myself" /
></td></tr>
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject:</label> <input id="id_f
or_subject" type="text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message:</label> <input type="te
xt" name="message" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender:</label> <input type="emai
l" name="sender" id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself:</label> <input type
="checkbox" name="cc_myself" id="id_for_cc_myself" /></li>
>>> print(f.as_p())
<p><label for="id_for_subject">Subject:</label> <input id="id_fo
r_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_for_message">Message:</label> <input type="tex
t" name="message" id="id_for_message" /></p>
<p><label for="id_for_sender">Sender:</label> <input type="email
" name="sender" id="id_for_sender" /></p>
<p><label for="id_for_cc_myself">Cc myself:</label> <input type=
"checkbox" name="cc_myself" id="id_for_cc_myself" /></p>
```

如果 `auto_id` 设置为任何其它的真值 —— 例如不包含 `%s` 的字符串 —— 那么其行为将类似 `auto_id` 等于 `True` 。

默认情况下， `auto_id` 设置为 `'id_%s'` 。

`Form.``label_suffix`

一个字符串（默认为英文的 `:` ），表单渲染时将附加在每个label 名称的后面。

使用 `label_suffix` 参数可以自定义这个字符，或者完全删除它：

```
>>> f = ContactForm(auto_id='id_for_%s', label_suffix='')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject</label> <input id="id_fo
r_subject" type="text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message</label> <input type="tex
t" name="message" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender</label> <input type="email
" name="sender" id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself</label> <input type=
"checkbox" name="cc_myself" id="id_for_cc_myself" /></li>
>>> f = ContactForm(auto_id='id_for_%s', label_suffix=' ->')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject -></label> <input id="id
_for_subject" type="text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message -></label> <input type="
text" name="message" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender -></label> <input type="em
ail" name="sender" id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself -></label> <input ty
pe="checkbox" name="cc_myself" id="id_for_cc_myself" /></li>
```

注意，该标签后缀只有当label 的最后一个字符不是表单符号（ `.` , `!` , `?` 和 `:` ）时才添加。

New in Django 1.8.

字段可以定义自己的 `label_suffix` 。而且将优先于 `Form.label_suffix` 。在运行时刻，后缀可以使用 `label_tag()` 的 `label_suffix` 参数覆盖。

字段的顺序

在 `as_p()` 、 `as_ul()` 和 `as_table()` 中，字段以表单类中定义的顺序显示。例如，在 `ContactForm` 示例中，字段定义的顺序为 `subject` , `message` , `sender` , `cc_myself` 。若要重新排序HTML 中的输出，只需改变字段在类中列出的顺序。

错误如何显示

如果你渲染一个绑定的 `表单` 对象，渲染时将自动运行表单的验证，HTML 输出将在出错字段的附近以 `<ul class="errorlist">` 形式包含验证的错误。错误信息的位置与你使用的输出方法有关：

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data, auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><ul class="errorlist"><li>This field is
 required.</li></ul><input type="text" name="subject" maxlength=
"100" /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" value
="Hi there" /></td></tr>
<tr><th>Sender:</th><td><ul class="errorlist"><li>Enter a valid
email address.</li></ul><input type="email" name="sender" value=
"invalid email address" /></td></tr>
<tr><th>Cc myself:</th><td><input checked="checked" type="checkb
ox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li><ul class="errorlist"><li>This field is required.</li></ul>S
ubject: <input type="text" name="subject" maxlength="100" /></li
>
<li>Message: <input type="text" name="message" value="Hi there"
/></li>
<li><ul class="errorlist"><li>Enter a valid email address.</li><
/ul>Sender: <input type="email" name="sender" value="invalid ema
il address" /></li>
<li>Cc myself: <input checked="checked" type="checkbox" name="cc
_myself" /></li>
>>> print(f.as_p())
<p><ul class="errorlist"><li>This field is required.</li></ul></
p>
<p>Subject: <input type="text" name="subject" maxlength="100" />
</p>
<p>Message: <input type="text" name="message" value="Hi there" /
></p>
<p><ul class="errorlist"><li>Enter a valid email address.</li></
ul></p>
<p>Sender: <input type="email" name="sender" value="invalid emai
l address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_
myself" /></p>
```

自定义错误清单的格式

默认情况下，表单使用 `django.forms.utils.ErrorList` 来格式化验证时的错误。如果你希望使用另外一种类来显示错误，可以在构造时传递（在Python 2 中将 `__str__` 替换为 `__unicode__` ）：

```
>>> from django.forms.utils import ErrorList
>>> class DivErrorList(ErrorList):
...     def __str__(self):              # __unicode__ on Python 2
...         return self.as_divs()
...     def as_divs(self):
...         if not self: return ''
...         return '<div class="errorlist">%s</div>' % ''.join(['<div class="error">%s</div>' % e for e in self])
>>> f = ContactForm(data, auto_id=False, error_class=DivErrorList)
>>> f.as_p()
<div class="errorlist"><div class="error">This field is required.</div></div>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<div class="errorlist"><div class="error">Enter a valid email address.</div></div>
<p>Sender: <input type="email" name="sender" value="invalid email address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

Changed in Django 1.7:

`django.forms.util` 重命名为 `django.forms.utils` 。

更细粒度的输出

`as_p()` 、 `as_ul()` 和 `as_table()` 方法是为懒惰的程序员准备的简单快捷方法 —— 它们不是显示表单的唯一方式。

class `BoundField`

用于显示HTML 表单或者访问 表单 实例的一个属性。

其 `__str__()` （Python 2 上为 `__unicode__` ）方法显示该字段的HTML。

以字段的名称为键，用字典查询语法查询表单，可以获取一个 `BoundField` ：

```
>>> form = ContactForm()
>>> print(form['subject'])
<input id="id_subject" type="text" name="subject" maxlength="100" />
```

迭代表单可以获取所有的 `BoundField` ：

```
>>> form = ContactForm()
>>> for boundfield in form: print(boundfield)
<input id="id_subject" type="text" name="subject" maxlength="100
" />
<input type="text" name="message" id="id_message" />
<input type="email" name="sender" id="id_sender" />
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

字段的输出与表单的 `auto_id` 设置有关：

```
>>> f = ContactForm(auto_id=False)
>>> print(f['message'])
<input type="text" name="message" />
>>> f = ContactForm(auto_id='id_%s')
>>> print(f['message'])
<input type="text" name="message" id="id_message" />
```

若要获取字段的错误列表，可以访问字段的 `errors` 属性。

`BoundField.``errors`

一个类列表对象，打印时以HTML `<ul class="errorlist">` 形式显示：

```
>>> data = {'subject': 'hi', 'message': '', 'sender': '', 'cc_my
self': ''}
>>> f = ContactForm(data, auto_id=False)
>>> print(f['message'])
<input type="text" name="message" />
>>> f['message'].errors
['This field is required.']
>>> print(f['message'].errors)
<ul class="errorlist"><li>This field is required.</li></ul>
>>> f['subject'].errors
[]
>>> print(f['subject'].errors)

>>> str(f['subject'].errors)
''
```

`BoundField.``label_tag` (*contents=None*, *attrs=None*, *label_suffix=None*)

可以调用 `label_tag` 方法单独渲染表单字段的label 标签：

```
>>> f = ContactForm(data)
>>> print(f['message'].label_tag())
<label for="id_message">Message:</label>
```

如果你提供一个可选的 `contents` 参数，它将替换自动生成的label 标签。另外一个可选的 `attrs` 参数可以包含 `<label>` 标签额外的属性。

生成的HTML 包含表单的 `label_suffix` （默认为一个冒号），或者当前字段的 `label_suffix` 。可选的 `label_suffix` 参数允许你覆盖之前设置的后缀。例如，你可以使用一个空字符串来隐藏已选择字段的label。如果在模板中需要这样做，你可以编写一个自定义的过滤器来允许传递参数给 `label_tag` 。

Changed in Django 1.8:

如果可用，label 将包含 `required_css_class` 。

`BoundField.``css_classes` ()

当你使用Django 的快捷的渲染方法时，习惯使用CSS 类型来表示必填的表单字段和有错误的字段。如果你是手工渲染一个表单，你可以使用 `css_classes` 方法访问这些CSS 类型：

```
>>> f = ContactForm(data)
>>> f['message'].css_classes()
'required'
```

除了错误和必填的类型之外，如果你还想提供额外的类型，你可以用参数传递它们：

```
>>> f = ContactForm(data)
>>> f['message'].css_classes('foo bar')
'foo bar required'
```

`BoundField.``value` ()

这个方法用于渲染字段的原始值，与用 `Widget` 渲染的值相同：

```
>>> initial = {'subject': 'welcome'}
>>> unbound_form = ContactForm(initial=initial)
>>> bound_form = ContactForm(data, initial=initial)
>>> print(unbound_form['subject'].value())
welcome
>>> print(bound_form['subject'].value())
hi
```

`BoundField.``id_for_label`

使用这个属性渲染字段的ID。例如，如果你在模板中手工构造一个 `<label>` （尽管 `label_tag()` 将为你这么做）：

```
<label for="{{ form.my_field.id_for_label }}">...</label>{{ my_field }}
```

默认情况下，它是在字段名称的前面加上 `id_` （上面的例子中将是“ `id_my_field` ”）。你可以通过设置字段Widget 的 `attrs` 来修改ID。例如，像这样声明一个字段：

```
my_field = forms.CharField(widget=forms.TextInput(attrs={'id': 'myFIELD'}))
```

使用上面的模板，将渲染成：

```
<label for="myFIELD">...</label><input id="myFIELD" type="text" name="my_field" />
```

绑定上传的文件到表单

处理带有 `FileField` 和 `ImageField` 字段的表单比普通的表单要稍微复杂一点。

首先，为了上传文件，你需要确保你的 `<form>` 元素正确定义 `enctype` 为 `"multipart/form-data"` ：

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

其次，当你使用表单时，你需要绑定文件数据。文件数据的处理与普通的表单数据是分开的，所以如果表单包含 `FileField` 和 `ImageField` ，绑定表单时你需要指定第二个参数。所以，如果我们扩展ContactForm 并包含一个名为 `mugshot` 的 `ImageField` ，我们需要绑定包含mugshot 图片的文件数据：

```
# Bound form with an image field
>>> from django.core.files.uploadedfile import SimpleUploadedFile
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> file_data = {'mugshot': SimpleUploadedFile('face.jpg', <file data>)}
>>> f = ContactFormWithMugshot(data, file_data)
```

实际上，你一般将使用 `request.FILES` 作为文件数据的源（和使用 `request.POST` 作为表单数据的源一样）：

```
# Bound form with an image field, data from the request
>>> f = ContactFormWithMugshot(request.POST, request.FILES)
```

构造一个未绑定的表单和往常一样 —— 将表单数据和文件数据同时省略：

```
# Unbound form with an image field
>>> f = ContactFormWithMugshot()
```

测试**multipart** 表单

`Form.``is_multipart` ()

如果你正在编写可重用的视图或模板，你可能事先不知道你的表单是否是一个multipart 表单。 `is_multipart()` 方法告诉你表单提交时是否要求multipart：

```
>>> f = ContactFormWithMugshot()
>>> f.is_multipart()
True
```

下面是如何在模板中使用它的一个示例：

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post" action="/f
oo/">
{% else %}
    <form method="post" action="/foo/">
{% endif %}
{{ form }}
</form>
```

子类化表单

如果你有多个 `表单` 类共享相同的字段，你可以使用子类化来减少冗余。

当你子类化一个自定义的 `表单` 类时，生成的子类将包含父类中的所有字段，以及在子类中定义的字段。

在下面的例子中， `ContactFormWithPriority` 包含 `ContactForm` 中的所有字段，以及另外一个字段 `priority` 。排在前面的是 `ContactForm` 中的字段：

```
>>> class ContactFormWithPriority(ContactForm):
...     priority = forms.CharField()
>>> f = ContactFormWithPriority(auto_id=False)
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" /
></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="email" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
<li>Priority: <input type="text" name="priority" /></li>
```

可以子类化多个表单，将表单作为“mix-ins”。在下面的例子中， `BeatleForm` 子类化 `PersonForm` 和 `InstrumentForm` ，所以它的字段列表包含两个父类的所有字段：

```
>>> from django.forms import Form
>>> class PersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
>>> class InstrumentForm(Form):
...     instrument = CharField()
>>> class BeatleForm(PersonForm, InstrumentForm):
...     haircut_type = CharField()
>>> b = BeatleForm(auto_id=False)
>>> print(b.as_ul())
<li>First name: <input type="text" name="first_name" /></li>
<li>Last name: <input type="text" name="last_name" /></li>
<li>Instrument: <input type="text" name="instrument" /></li>
<li>Haircut type: <input type="text" name="haircut_type" /></li>
```

New in Django 1.7.

- 在子类中，可以通过设置名字为 `None` 来删除从父类中继承的 `字段` 。例如：

```
>>> from django import forms

>>> class ParentForm(forms.Form):
...     name = forms.CharField()
...     age = forms.IntegerField()

>>> class ChildForm(ParentForm):
...     name = None

>>> ChildForm().fields.keys()
... ['age']
```

表单前缀

`Form.``prefix`

你可以将几个Django 表单放在一个 `<form>` 标签中。为了给每个 `表单` 一个自己的命名空间，可以使用 `prefix` 关键字参数：

```
>>> mother = PersonForm(prefix="mother")
>>> father = PersonForm(prefix="father")
>>> print(mother.as_ul())
<li><label for="id_mother-first_name">First name:</label> <input
 type="text" name="mother-first_name" id="id_mother-first_name"
/></li>
<li><label for="id_mother-last_name">Last name:</label> <input t
ype="text" name="mother-last_name" id="id_mother-last_name" /></
li>
>>> print(father.as_ul())
<li><label for="id_father-first_name">First name:</label> <input
 type="text" name="father-first_name" id="id_father-first_name"
/></li>
<li><label for="id_father-last_name">Last name:</label> <input t
ype="text" name="father-last_name" id="id_father-last_name" /></
li>
```

表单字段

class `Field` (***kwargs*)

创建一个 `表单` 类时，最重要的部分是定义表单的字段。每个字段都可以有自定义的验证逻辑，以及一些其它的钩子。

`Field.``clean` (*value*)

虽然 `字段` 类主要使用在 `表单` 类中，但你也可以直接实例化它们来使用，以便更好地了解它们是如何工作的。每个 `字段` 实例都有一个 `clean()` 方法， 它接受一个参数，然后返回“清洁的”数据或者抛出一个 `django.forms.ValidationError` 异常：

```
>>> from django import forms
>>> f = forms.EmailField()
>>> f.clean('foo@example.com')
'foo@example.com'
>>> f.clean('invalid email address')
Traceback (most recent call last):
...
ValidationError: ['Enter a valid email address.']
```

字段的核心参数

每个 `字段` 类的构造函数至少接受这些参数。有些 `字段` 类接受额外的、字段特有的参数，但以下参数应该总是能接受：

需要

`Field.``required`

默认情况下，每个 `字段` 类都假设必需有值，所以如果你传递一个空的值 —— 不管是 `None` 还是空字符串(`""`) —— `clean()` 将引发一个 `ValidationError` 异常：

```
>>> from django import forms
>>> f = forms.CharField()
>>> f.clean('foo')
'foo'
>>> f.clean('')
Traceback (most recent call last):
...
ValidationError: ['This field is required.']
>>> f.clean(None)
Traceback (most recent call last):
...
ValidationError: ['This field is required.']
>>> f.clean(' ')
' '
>>> f.clean(0)
'0'
>>> f.clean(True)
'True'
>>> f.clean(False)
'False'
```

若要表示一个字段不是必需的，请传递 `required=False` 给 `字段` 的构造函数：

```
>>> f = forms.CharField(required=False)
>>> f.clean('foo')
'foo'
>>> f.clean('')
''
>>> f.clean(None)
''
>>> f.clean(0)
'0'
>>> f.clean(True)
'True'
>>> f.clean(False)
'False'
```

如果 `字段` 具有 `required=False` ，而你传递给 `clean()` 一个空值， `clean()` 将返回一个转换后的空值而不是引发 `ValidationError` 。例如 `CharField` ，它将是一个空的Unicode 字符串。对于其它 `字段` 类，它可能是 `None` 。（每个字段各不相同）。

标签

`Field.``label`

`label` 参数让你指定字段“对人类友好”的label。当 `字段` 在 `表单` 中显示时将用到它。

正如在前面“输出表单为HTML”中解释的， `字段` 默认label 是通过将字段名中所有的下划线转换成空格并大写第一个字母生成的。如果默认的标签不合适，可以指定 `label` 。

下面是一个完整示例， `表单` 为它的两个字段实现了 `label` 。我们指定 `auto_id=False` 来让输出简单一些：

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(label='Your name')
...     url = forms.URLField(label='Your Web site', required=Fal
se)
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Your name:</th><td><input type="text" name="name" /></td
></tr>
<tr><th>Your Web site:</th><td><input type="url" name="url" /></
td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></t
d></tr>
```

label_suffix

`Field.``label_suffix`

New in Django 1.8.

`label_suffix` 参数让你基于每个字段覆盖表单的 `label_suffix` ：

```
>>> class ContactForm(forms.Form):
...     age = forms.IntegerField()
...     nationality = forms.CharField()
...     captcha_answer = forms.IntegerField(label='2 + 2', label
_suffix=' =')
>>> f = ContactForm(label_suffix='?')
>>> print(f.as_p())
<p><label for="id_age">Age?</label> <input id="id_age" name="age
" type="number" /></p>
<p><label for="id_nationality">Nationality?</label> <input id="i
d_nationality" name="nationality" type="text" /></p>
<p><label for="id_captcha_answer">2 + 2 =</label> <input id="id_
captcha_answer" name="captcha_answer" type="number" /></p>
```

unindent does not match any outer indentation level

`Field.``initial`

`initial` 参数让你指定渲染未绑定的 表单 中的 字段 时使用的初始值。

若要指定动态的初始数据，参见 `Form.initial` 参数。

这个参数的使用场景是当你想要显示一个“空”的表单，其某个字段初始化为一个特定的值。例如：

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

你可能正在想为什么不在显示表单的时候传递一个包含初始化值的字典？如果这么做，你将触发验证过程，此时HTML 输出将包含任何验证中产生的错误：

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField()
...     url = forms.URLField()
...     comment = forms.CharField()
>>> default_data = {'name': 'Your name', 'url': 'http://'}
>>> f = CommentForm(default_data, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" /></td></tr>
<tr><th>Url:</th><td><ul class="errorlist"><li>Enter a valid URL.</li></ul><input type="url" name="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td><ul class="errorlist"><li>This field is required.</li></ul><input type="text" name="comment" /></td></tr>
```

这就是为什么 `initial` 的值只在未绑定的表单中显示的原因。对于绑定的表单，HTML 输出将使用绑定的数据。

还要注意，如果某个字段的值没有给出， `initial` 值不会作为“后备”的数据。 `initial` 值只用于原始表单的显示：

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> data = {'name': '', 'url': '', 'comment': 'Foo'}
>>> f = CommentForm(data)
>>> f.is_valid()
False
# The form does *not* fall back to using the initial values.
>>> f.errors
{'url': ['This field is required.'], 'name': ['This field is req
uired.']}
```

除了常数之外，你还可以传递一个可调用的对象：

```
>>> import datetime
>>> class DateForm(forms.Form):
...     day = forms.DateField(initial=datetime.date.today)
>>> print(DateForm())
<tr><th>Day:</th><td><input type="text" name="day" value="12/23/
2008" /><td></tr>
```

可调用对象在未绑定的表单显示的时候才计算，不是在定义的时候。

窗口小部件

`Field.``widget`

`widget` 参数让你指定渲染 `表单` 时使用的 `Widget` 类。更多信息参见 Widgets。

help_text

`Field.``help_text`

`help_text` 参数让你指定 `字段` 的描述文本。如果提供 `help_text` ，在通过 `表单` 的便捷方法（例如， `as_ul()` ）渲染 `字段` 时，它将紧接着 `字段` 显示。

下面是一个完整的示例， `表单` 为它的两个字段实现了 `help_text` 。 我们指定 `auto_id=False` 来让输出简单一些：

```
>>> from django import forms
>>> class HelpTextContactForm(forms.Form):
...     subject = forms.CharField(max_length=100, help_text='100
 characters max.')
...     message = forms.CharField()
...     sender = forms.EmailField(help_text='A valid email addre
ss, please.')
...     cc_myself = forms.BooleanField(required=False)
>>> f = HelpTextContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxle
ngth="100" /><br /><span class="helptext">100 characters max.</s
pan></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" /></t
d></tr>
<tr><th>Sender:</th><td><input type="email" name="sender" /><br
/>A valid email address, please.</td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_mysel
f" /></td></tr>
>>> print(f.as_ul()))
<li>Subject: <input type="text" name="subject" maxlength="100" /
> <span class="helptext">100 characters max.</span></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="email" name="sender" /> A valid email a
ddress, please.</li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" />
 <span class="helptext">100 characters max.</span></p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="email" name="sender" /> A valid email ad
dress, please.</p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

error_messages

`Field.``error_messages`

`error_messages` 参数让你覆盖字段引发的异常中的默认信息。传递的是一个字典，其键为你想覆盖的错误信息。例如，下面是默认的错误信息：

```
>>> from django import forms
>>> generic = forms.CharField()
>>> generic.clean('')
Traceback (most recent call last):
  ...
ValidationError: ['This field is required.']
```

而下面是自定义的错误信息：

```
>>> name = forms.CharField(error_messages={'required': 'Please e
nter your name'})
>>> name.clean('')
Traceback (most recent call last):
  ...
ValidationError: ['Please enter your name']
```

在下面的内建的字段一节中，每个 `字段` 都定义了它自己的错误信息。

验证器

`Field.``validators`

`validators` 参数让你可以为字段提供一个验证函数的列表。

更多的信息，参见*validators* 文档 。

本地化

`Field.``localize`

`localize` 参数启用表单数据的本地化，包括输入和输出。

更多信息，参见本地化格式。

检查字段数据是否已经改变

已经改变**()**

`Field.``has_changed` ()

Changed in Django 1.8:

重命名 `_has_changed()` 为该方法。

`has_changed()` 方法用于决定字段的值是否从初始值发生了改变。返回 `True` 或 `False` 。

更多信息，参见 `Form.has_changed()` 。

内建的 `字段`

自然， `表单` 的库会带有一系列表示常见需求的 `字段` 。 这一节记录每个内建字段。

对于每个字段，我们描述默认的 `widget` 。我们还会指出提供空值时的返回值（参见上文的 `required` 以理解它的含义）。

BooleanField

class `BooleanField` (***kwargs*)

- 默认的Widget： `CheckboxInput`
- 空值： `False`
- 规范化为：Python 的 `True` 或 `False` 。
- 如果字段带有 `required=True` ，验证值是否为 `True` （例如复选框被勾上）。
- 错误信息的键： `required`

注

因为所有的 `Field` 子类都默认带有 `required=True` ，这里的验证条件很重要。如果你希望表单中包含一个既可以为 `True` 也可以为 `False` 的布尔值（例如，复选框可以勾上也可以不勾上），你必须要记住在创建 `BooleanField` 时传递 `required=False` 。

CharField

class `CharField` (***kwargs*)

- 默认的Widget： `TextInput`
- 空值： `''` （一个空字符串）
- 规范化为：一个Unicode 对象。
- 如果提供，验证 `max_length` 或 `min_length` 。 否则，所有的输入都是合法的。
- 错误信息的键： `required` , `max_length` , `min_length`

有两个参数用于验证：

`max_length`

`min_length`

如果提供，这两个参数将确保字符串的最大和最小长度。

选择字段

class `ChoiceField` (***kwargs*)

- 默认的Widget： `Select`
- 空值： `''` （一个空字符串）
- 规范化为：一个Unicode 对象。
- 验证给定的值在选项列表中存在。

- 错误信息的键：`required`，`invalid_choice`

`invalid_choice` 错误消息可能包含 `%(value)s`，它将被选择的选项替换掉。

接收一个额外的必选参数：

`choices`

用来作为该字段选项的一个二元组组成的可迭代对象（例如，列表或元组）或者一个可调用对象。参数的格式与模型字段的 `choices` 参数相同。更多细节参见模型字段参考中关于选项的文档。如果参数是可调用的，它在字段的表单初始化时求值。

Changed in Django 1.8:

添加传递一个可调用对象给 `choices` 的功能。

TypedChoiceField

class `TypedChoiceField` (***kwargs*)

与 `ChoiceField` 很像，只是 `TypedChoiceField` 接受两个额外的参数 `coerce` 和 `empty_value`。

- 默认的Widget：`Select`
- 空值：`empty_value`
- 规范化为：`coerce` 参数类型的值。
- 验证给定的值在选项列表中存在并且可以被强制转换。
- 错误信息的键：`required`，`invalid_choice`

接收的额外参数：

`coerce`

接收一个参数并返回强制转换后的值的一个函数。例如内建的 `int`、`float`、`bool` 和其它类型。默认为id 函数。注意强制转换在输入验证结束后发生，所以它可能强制转换不在 `choices` 中的值。

`empty_value`

用于表示“空”的值。默认为空字符串；`None` 是另外一个常见的选项。注意这个值不会被 `coerce` 参数中指定的函数强制转换，所以请根据情况进行选择。

DateField

class `DateField` (***kwargs*)

- 默认的Widget：`DateInput`
- 空值：`None`
- 规范化为：一个Python `datetime.date` 对象。
- 验证给出的值是一个 `datetime.date`、`datetime.datetime` 或指定日期

格式的字符串。
- 错误信息的键：`required`，`invalid`

接收一个可选的参数：

`input_formats`

一个格式的列表，用于转换一个字符串为 `datetime.date` 对象。

如果没有提供 `input_formats`，默认的输入格式为：

```
['%Y-%m-%d',       # '2006-10-25'
'%m/%d/%Y',        # '10/25/2006'
'%m/%d/%y']        # '10/25/06'
```

另外，如果你在设置中指定 `USE_L10N=False`，以下的格式也将包含在默认的输入格式中：

```
['%b %d %Y',       # 'Oct 25 2006'
'%b %d, %Y',       # 'Oct 25, 2006'
'%d %b %Y',        # '25 Oct 2006'
'%d %b, %Y',       # '25 Oct, 2006'
'%B %d %Y',        # 'October 25 2006'
'%B %d, %Y',       # 'October 25, 2006'
'%d %B %Y',        # '25 October 2006'
'%d %B, %Y']       # '25 October, 2006'
```

另见本地化格式。

DateTimeField

class `DateTimeField` (***kwargs*)

- 默认的Widget：`DateTimeInput`
- 空值：`None`
- 规范化为：一个Python `datetime.datetime` 对象。
- 验证给出的值是一个 `datetime.date`、`datetime.datetime` 或指定日期格式的字符串。
- 错误信息的键：`required`，`invalid`

接收一个可选的参数：

`input_formats`

一个格式的列表，用于转换一个字符串为 `datetime.datetime` 对象。

如果没有提供 `input_formats`，默认的输入格式为：

```
['%Y-%m-%d %H:%M:%S',     # '2006-10-25 14:30:59'
'%Y-%m-%d %H:%M',         # '2006-10-25 14:30'
'%Y-%m-%d',               # '2006-10-25'
'%m/%d/%Y %H:%M:%S',      # '10/25/2006 14:30:59'
'%m/%d/%Y %H:%M',         # '10/25/2006 14:30'
'%m/%d/%Y',               # '10/25/2006'
'%m/%d/%y %H:%M:%S',      # '10/25/06 14:30:59'
'%m/%d/%y %H:%M',         # '10/25/06 14:30'
'%m/%d/%y']               # '10/25/06'
```

另见本地化格式。

自1.7版起已弃用： `DateTimeField` 与 `SplitDateTimeWidget` 一起使用的功能被废弃并将在Django 1.9 中删除。请改用 `SplitDateTimeField` 。

DecimalField

class `DecimalField` (***kwargs*)

- 默认的Widget：当 `Field.localize` 是 `False` 时为 `NumberInput` ，否则为 `TextInput` 。
- 空值： `None`
- 规范化为：一个Python `decimal` 。
- 验证给定的值为一个十进制数。忽略前导和尾随的空白。
- 错误信息的键： `required` , `invalid` , `max_value` , `min_value` , `max_digits` , `max_decimal_places` , `max_whole_digits`

`max_value` 和 `min_value` 错误信息可能包含 `%(limit_value)s` ，它们将被真正的限制值替换。类似地， `max_digits` 、 `max_decimal_places` 和 `max_whole_digits` 错误消息可能包含 `%(max)s` 。

接收四个可选的参数：

`max_value`

`min_value`

它们控制字段中允许的值的范围，应该以 `decimal.Decimal` 值给出。

`max_digits`

值允许的最大位数（小数点之前和之后的数字总共的位数，前导的零将被删除）。

`decimal_places`

允许的最大小数位。

DurationField

New in Django 1.8.

class `DurationField` (***kwargs*)

- 默认的Widget： `TextInput`
- 空值： `None`
- 规范化为：一个Python `timedelta` 。
- 验证给出的值是一个字符串，而可以给转换为 `timedelta` 。
- 错误信息的键： `required` , `invalid` .

接收任何可以被 `parse_duration()` 理解的格式。

EmailField

class `EmailField` (***kwargs*)

- 默认的Widget： `EmailInput`
- 空值： `''` （一个空字符串）
- 规范化为：一个Unicode 对象。
- 验证给出的值是一个合法的邮件地址，使用一个适度复杂的正则表达式。
- 错误信息的键： `required` , `invalid`

具有两个可选的参数用于验证， `max_length` 和 `min_length` 。如果提供，这两个参数确保字符串的最大和最小长度。

FileField

class `FileField` (***kwargs*)

- 默认的Widget： `ClearableFileInput`
- 空值： `None`
- 规范化为：一个 `UploadedFile` 对象，它封装文件内容和文件名为一个单独的对象。
- 可以验证非空的文件数据已经绑定到表单。
- 错误信息的键： `required` , `invalid` , `missing` , `empty` , `max_length`

具有两个可选的参数用于验证， `max_length` 和 `allow_empty_file` 。如果提供，这两个参数确保文件名的最大长度，而且即使文件内容为空时验证也会成功。

若要了解 `UploadedFile` 对象的更多内容，参见文件上传的文档。

当你在表单中使用 `FileField` 时，必须要记住绑定文件数据到表单上。

`max_length` 错误信息表示文件名的长度。在错误信息中， `%(max)d` 将替换为文件的最大长度， `%(length)d` 将替换为当前文件名的长度。

FilePathField

class `FilePathField` (***kwargs*)

- 默认的Widget：`Select`
- 空值：`None`
- 规范化为：一个Unicode 对象。
- 验证选择的选项在选项列表中存在。
- 错误信息的键：`required`, `invalid_choice`

这个字段允许从一个特定的目录选择文件。它接受三个额外的参数；只有`path`是必需的：

`path`

你想要列出的目录的绝对路径。这个目录必须存在。

`recursive`

如果为`False`（默认值），只用直接位于`path`下的文件或目录作为选项。如果为`True`，将递归访问这个目录，其所有的子目录和文件都将作为选项。

`match`

正则表达式表示的一个模式；只有匹配这个表达式的名称才允许作为选项。

`allow_files`

可选。为`True`或`False`。默认为`True`。表示是否应该包含指定位置的文件。它和`allow_folders`必须有一个为`True`。

`allow_folders`

可选。为`True`或`False`。默认为`False`。表示是否应该包含指定位置的目录。它和`allow_files`必须有一个为`True`。

FloatField

class `FloatField`(***kwargs*)

- 默认的Widget：当`Field.localize`是`False`时为`NumberInput`，否则为`TextInput`。
- 空值：`None`
- 规范化为：一个Float 对象。
- 验证给出的值是一个浮点数。和Python 的`float()`函数一样，允许前导和尾随的空白符。
- 错误信息的键：`required`, `invalid`, `max_value`, `min_value`

接收两个可选的参数用于验证，`max_value`和`min_value`。它们控制字段中允许的值的范围。

ImageField

class `ImageField`(***kwargs*)

- 默认的Widget： `ClearableFileInput`
- 空值： `None`
- 规范化为: An `UploadedFile` object that wraps the file content and file name into a single object.
- 验证文件数据已绑定到表单，并且该文件具有Pillow理解的图像格式。
- 错误信息的键： `required` , `invalid` , `missing` , `empty` , `invalid_image`

使用 `ImageField` 需要安装Pillow并支持您使用的图像格式。如果在上传图片时遇到 `损坏 图像` 错误，通常意味着Pillow不了解其格式。要解决这个问题，请安装相应的库并重新安装Pillow。

在表单上使用 `ImageField` 时，您还必须记住*bind the file data to the form*。

Changed in Django 1.8:

在字段清理和验证后， `UploadedFile` 对象将有一个额外的 `image` 属性，包含Pillow 图像实例，用于检查文件是一个有效的图像。 `UploadedFile.content_type` 也会更新为由Pillow确定的图片内容类型。

IntegerField

class `IntegerField` (***kwargs*)

- 默认的Widget：当 `Field.localize` 是 `False` 时为 `NumberInput` ，否则为 `TextInput` 。
- 空值： `None`
- 规范化为：一个Python 整数或长整数。
- 验证给定值是一个整数。允许前导和尾随空格，如Python的 `int()` 函数。
- 错误信息的键： `required` , `invalid` , `max_value` , `min_value`

`max_value` 和 `min_value` 错误消息可能包含 `%(limit_value)s` ，将替换为适当的限制。

采用两个可选参数进行验证：

`max_value`

`min_value`

它们控制字段中允许的值的范围。

IPAddressField

class `IPAddressField` (***kwargs*)

自1.7版起已弃用：此字段已弃用，支持 `GenericIPAddressField` 。

- 默认的Widget： `TextInput`
- 空值： `''` （一个空字符串）

- 规范化为：一个Unicode 对象。
- 使用正则表达式验证给定值是有效的IPv4地址。
- 错误信息的键：`required`, `invalid`

GenericIPAddressField

class `GenericIPAddressField` (***kwargs*)

包含IPv4或IPv6地址的字段。

- 默认的Widget：`TextInput`
- 空值：`''`（一个空字符串）
- 规范化为：一个Unicode 对象。IPv6地址如下所述进行归一化。
- 验证给定值是有效的IP地址。
- 错误信息的键：`required`, `invalid`

IPv6地址规范化遵循 **RFC 4291**部分2.2，包括使用该部分第3段中建议的IPv4格式，如 `::ffff:192.0.2.0` 例如，`2001:0::0:01` 将被标准化为 `2001::1` 和 `::ffff:0a0a:0a0a` `::ffff:10.10.10.10` 。所有字符都转换为小写。

有两个可选参数：

`protocol`

限制指定协议的有效输入。接受的值为 `both` （默认值），`IPv4` 或 `IPv6` 。匹配不区分大小写。

`unpack_ipv4`

解开IPv4映射地址，例如 `::ffff:192.0.2.1` 。如果启用此选项，则该地址将解包到 `192.0.2.1` 。默认为禁用。只能在 `protocol` 设置为 `'both'` 时使用。

MultipleChoiceField

class `MultipleChoiceField` (***kwargs*)

- 默认的Widget：`SelectMultiple`
- 空值：`[]`（一个空列表）
- 规范化为：一个Unicode 对象列表。
- 验证给定值列表中的每个值都存在于选择列表中。
- 错误信息的键：`required`, `invalid_choice`, `invalid_list`

`invalid_choice` 错误消息可能包含 `%(value)s` ，将替换为所选择的选项。

对于 `ChoiceField` ，需要一个额外的必需参数 `choices` 。

TypedMultipleChoiceField

class `TypedMultipleChoiceField` (***kwargs*)

就像 `MultipleChoiceField` ，除了 `TypedMultipleChoiceField` 需要两个额外的参数， `coerce` 和 `empty_value` 。

- 默认的Widget： `SelectMultiple`
- 空值： `empty_value`
- 规范化为： `coerce` 参数提供的类型值列表。
- 验证给定值存在于选项列表中并且可以强制。
- 错误信息的键： `required` , `invalid_choice`

`invalid_choice` 错误消息可能包含 `%(value)s` ，将替换为所选择的选项。

对于 `TypedChoiceField` ，需要两个额外的参数 `coerce` 和 `empty_value` 。

NullBooleanField

class `NullBooleanField` (***kwargs*)

- 默认的Widget： `NullBooleanSelect`
- 空值： `None`
- 规范化为：一个Python `True` , `False` 或 `None` 值。
- 不验证任何内容（即，它从不引发 `ValidationError` ）。

RegexField

class `RegexField` (***kwargs*)

- 默认的Widget： `TextInput`
- 空值： `''` （一个空字符串）
- 规范化为：一个Unicode对象
- 验证给定值与某个正则表达式匹配。
- 错误信息的键： `required` , `invalid`

需要一个必需的参数：

`regex`

指定为字符串或编译的正则表达式对象的正则表达式。

也采用 `max_length` 和 `min_length` ，它们的作用与 `CharField` 相同。

自1.8版起已弃用：可接受的参数 `error_message` 也被接受用于向后兼容性，但会在Django 2.0中删除。提供错误消息的首选方法是使用 `error_messages` 参数，传递具有 `'invalid'` 的字典作为键，并将错误消息作为值。

SlugField

class `SlugField` (***kwargs*)

- 默认的Widget：`TextInput`
- 空值：`''`（一个空字符串）
- 规范化为：一个Unicode 对象。
- 验证给定的字符串只包括字母、数字、下划线及连字符。
- 错误信息的键：`required`, `invalid`

此字段用于在表单中表示模型 `SlugField`。

TimeField

class `TimeField` (***kwargs*)

- 默认的Widget：`TextInput`
- 空值：`None`
- 规范化为：一个Python 的 `datetime.time` 对象。
- 验证给定值是 `datetime.time` 或以特定时间格式格式化的字符串。
- 错误信息的键：`required`, `invalid`

采用一个可选参数：

`input_formats`

用于尝试将字符串转换为有效的 `datetime.time` 对象的格式列表。

如果未提供 `input_formats` 参数，则默认输入格式为：

```
'%H:%M:%S',     # '14:30:59'
'%H:%M',        # '14:30'
```

URLField

class `URLField` (***kwargs*)

- 默认的Widget：`URLInput`
- 空值：`''`（一个空字符串）
- 规范化为：一个Unicode 对象。
- 验证给定值是有效的URL。
- 错误信息的键：`required`, `invalid`

采用以下可选参数：

`max_length`

`min_length`

这些与 `CharField.max_length` 和 `CharField.min_length` 相同。

UUIDField

New in Django 1.8.

class `UUIDField` (***kwargs*)

- 默认的Widget： `TextInput`
- 空值： `''` （一个空字符串）
- 规范化为：一个 `UUID` 对象。
- 错误信息的键： `required` , `invalid`

此字段将接受任何作为 `UUID` 构造函数的 `hex` 参数接受的字符串格式。

稍微复杂一点的内建 `Field` 类

ComboField

class `ComboField` (***kwargs*)

- 默认的Widget： `TextInput`
- 空值： `''` （一个空字符串）
- 规范化为：一个Unicode 对象。
- 根据指定为 `ComboField` 的参数的每个字段验证给定值。
- 错误信息的键： `required` , `invalid`

需要一个额外的必需参数：

`fields`

应用于验证字段值的字段列表（按提供它们的顺序）。

```
&gt;&gt;&gt; from django.forms import ComboField
&gt;&gt;&gt; f = ComboField(fields=[CharField(max_length=20), Em
ailField()])
&gt;&gt;&gt; f.clean('test@example.com')
'test@example.com'
&gt;&gt;&gt; f.clean('longemailaddress@example.com')
Traceback (most recent call last):
...
ValidationError: ['Ensure this value has at most 20 characters (
it has 28).']
```

MultiValueField

class `MultiValueField` (*fields=(), **kwargs*)

- 默认的Widget： `TextInput`
- 空值： `''` （一个空字符串）
- 规范化为：子类的 `compress` 方法返回的类型。

- 针对指定为 `MultiValueField` 的参数的每个字段验证给定值。
- 错误信息的键：`required`，`invalid`，`incomplete`

聚合共同产生单个值的多个字段的逻辑。

此字段是抽象的，必须是子类。与单值字段相反，`MultiValueField` 的子类不能实现 `clean()`，而是实现 `compress()`。

需要一个额外的必需参数：

`fields`

字段的元组，其值被清除并随后组合成单个值。每个字段的值由 `字段中的相应字段清除` - 第一个值由第一个字段清除，第二个值由第二个字段清除等。清除所有字段后，通过 `compress()` 将干净值列表合并为一个值。

还需要一个额外的可选参数：

`require_all_fields`

New in Django 1.7.

默认为 `True`，在这种情况下，如果没有为任何字段提供值，则会出现 `required` 验证错误。

设置为 `False` 时，可以将 `Field.required` 属性设置为 `False`，以使其为可选字段。如果没有为必填字段提供值，则会出现 `不完整` 验证错误。

可以在 `MultiValueField` 子类上定义默认 `不完整` 错误消息，或者可以在每个单独字段上定义不同的消息。例如：

```
from django.core.validators import RegexValidator

class PhoneField(MultiValueField):
    def __init__(self, *args, **kwargs):
        # Define one message for all fields.
        error_messages = {
            'incomplete': 'Enter a country calling code and a ph
one number.',
        }
        # Or define a different message for each field.
        fields = (
            CharField(error_messages={'incomplete': 'Enter a cou
ntry calling code.'},
                      validators=[RegexValidator(r'^[0-9]+$', 'E
nter a valid country calling code.')]),
            CharField(error_messages={'incomplete': 'Enter a pho
ne number.'},
                      validators=[RegexValidator(r'^[0-9]+$', 'E
nter a valid phone number.')]),
            CharField(validators=[RegexValidator(r'^[0-9]+$', 'E
nter a valid extension.')],
                      required=False),
        )
        super(PhoneField, self).__init__(
            error_messages=error_messages, fields=fields,
            require_all_fields=False, *args, **kwargs)
```

`widget`

必须是 `django.forms.MultiWidget` 的子类。默认值为 `TextInput` ，在这种情况下可能不是非常有用。

`compress` (*data_list*)

获取有效值的列表，并在单个值中返回这些值的“压缩”版本。例如， `SplitDateTimeField` 是将时间字段和日期字段合并为 `datetime` 对象的子类。

此方法必须在子类中实现。

SplitDateTimeField

class `SplitDateTimeField` (***kwargs*)

- 默认的Widget： `SplitDateTimeWidget`
- 空值： `None`
- 规范化为：一个Python `datetime.datetime` 对象。
- 验证给定的值是 `datetime.datetime` 或以特定日期时间格式格式化的字符串。

- 错误信息的键：`required`，`invalid`，`invalid_date`，`invalid_time`

有两个可选参数：

`input_date_formats`

用于尝试将字符串转换为有效的 `datetime.date` 对象的格式列表。

如果未提供 `input_date_formats` 参数，则会使用 `DateField` 的默认输入格式。

`input_time_formats`

用于尝试将字符串转换为有效的 `datetime.time` 对象的格式列表。

如果未提供 `input_time_formats` 参数，则使用 `TimeField` 的默认输入格式。

处理关系的字段

两个字段可用于表示模型之间的关系：`ModelChoiceField` 和 `ModelMultipleChoiceField` 。这两个字段都需要单个 `queryset` 参数，用于创建字段的选择。在表单验证时，这些字段将把一个模型对象（在 `ModelChoiceField` 的情况下）或多个模型对象（在 `ModelMultipleChoiceField` 的情况下）放置到 `cleaned_data` 表单的字典。

对于更复杂的用法，可以在声明表单字段时指定 `queryset=None` ，然后在窗体的 `__init__()` 方法中填充 `queryset`

```
class FooMultipleChoiceForm(forms.Form):
    foo_select = forms.ModelMultipleChoiceField(queryset=None)

    def __init__(self, *args, **kwargs):
        super(FooMultipleChoiceForm, self).__init__(*args, **kwa
rgs)
        self.fields['foo_select'].queryset = ...
```

ModelChoiceField

class `ModelChoiceField` (***kwargs*)

- 默认的Widget：`Select`
- 空值：`None`
- 规范化为：一个模型实例。
- 验证给定的id存在于查询集中。
- 错误信息的键：`required`，`invalid_choice`

可以选择一个单独的模型对像，适用于表示一个外键字段。 `ModelChoiceField` 默认widet不适用选择数量很大的情况，在大于100项时应该避免使用它。

需要单个参数：

`queryset`

将导出字段选择的模型对象的 `QuerySet` ，将用于验证用户的选择。

`ModelChoiceField` 也有两个可选参数：

`empty_label`

默认情况下， `ModelChoiceField` 使用的 <select> 小部件将在列表顶部有一个空选项。您可以使用 `empty_label` 属性更改此标签的文本（默认为 `"---------"` ），也可以禁用空白标签完全通过将 `empty_label` 设置为 `None` ：

```
# A custom empty label
field1 = forms.ModelChoiceField(queryset=..., empty_label="(Noth
ing)")

# No empty label
field2 = forms.ModelChoiceField(queryset=..., empty_label=None)
```

请注意，如果需要 `ModelChoiceField` 并且具有默认初始值，则不会创建空选项（不管 `empty_label` 的值）。

`to_field_name`

此可选参数用于指定要用作字段窗口小部件中选项的值的字段。确保它是模型的唯一字段，否则选定的值可以匹配多个对象。默认情况下，它设置为 `None` ，在这种情况下，将使用每个对象的主键。例如：

```
# No custom to_field_name
field1 = forms.ModelChoiceField(queryset=...)
```

将产生：

```
&lt;select id="id_field1" name="field1"&gt;
&lt;option value="obj1.pk"&gt;Object1&lt;/option&gt;
&lt;option value="obj2.pk"&gt;Object2&lt;/option&gt;
...
&lt;/select&gt;
```

和：

```
# to_field_name provided
field2 = forms.ModelChoiceField(queryset=..., to_field_name="nam
e")
```

将产生：

```
&lt;select id="id_field2" name="field2"&gt;
&lt;option value="obj1.name"&gt;Object1&lt;/option&gt;
&lt;option value="obj2.name"&gt;Object2&lt;/option&gt;
...
&lt;/select&gt;
```

The `__str__` (`__unicode__` on Python 2) method of the model will be called to generate string representations of the objects for use in the field's choices; 提供自定义表示，子类 `ModelChoiceField` 并覆盖 `label_from_instance` 。此方法将接收一个模型对象，并应返回一个适合表示它的字符串。例如：

```
from django.forms import ModelChoiceField

class MyModelChoiceField(ModelChoiceField):
    def label_from_instance(self, obj):
        return "My Object #%i" % obj.id
```

ModelMultipleChoiceField

class `ModelMultipleChoiceField` (***kwargs*)

- 默认的Widget： `SelectMultiple`
- 空值： `QuerySet` (self.queryset.none())
- 规范化为： 模型实例的一个 `QuerySet` 。
- 验证在给定的值列表中的每个id存在于查询集中。
- 错误信息的键： `required` , `list` , `invalid_choice` , `invalid_pk_value`

`invalid_choice` 消息可以包含 `%(value)s` 并且 `invalid_pk_value` 消息可以包含 `%(pk)s` 其将被适当的值代替。

允许选择适合于表示多对多关系的一个或多个模型对象。与 `ModelChoiceField` 一样，您可以使用 `label_from_instance` 自定义对象表示， `queryset` 是必需的参数：

`queryset`

将导出字段选择的模型对象的 `QuerySet` ，将用于验证用户的选择。

创建自定义的字段

如果内建的 `字段` 不能满足你的需求，你可以很容易地创建自定义的 `字段` 。你需要创建 `django.forms.Field` 的一个子类。它只要求实现一个 `clean()` 方法和接收上面核心参数的 `__init__()` 方法(`required` , `label` , `initial` , `widget` , `help_text`)。

Widgets

Widget 是Django 对HTML 输入元素的表示。Widget 负责渲染HTML和提取GET/POST 字典中的数据。

小贴士

不要将Widget 与表单字段搞混淆。表单字段负责验证输入并直接在模板中使用。Widget 负责渲染网页上HTML 表单的输入元素和提取提交的原始数据。但是，Widget 需要赋值给表单的字段。

指定Widget

每当你指定表单的一个字段的时候，Django 将使用适合其数据类型的默认Widget。若要查找每个字段使用的Widget，参见内建的字段文档。

然而，如果你想要使用一个不同的Widget，你可以在定义字段时使用 `widget` 参数。例如：

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

这将使用一个 `Textarea` Widget来设置表单的评论 ，而不是默认的 `TextInput` Widget。

设置Widget 的参数

很多Widget 都有可选的参数；它们可以在定义字段的Widget 时设置。在下面的示例中，设置了 `SelectDateWidget` 的 `years` 属性：

```
from django import forms
from django.forms.extras.widgets import SelectDateWidget

BIRTH_YEAR_CHOICES = ('1980', '1981', '1982')
FAVORITE_COLORS_CHOICES = (('blue', 'Blue'),
                            ('green', 'Green'),
                            ('black', 'Black'))

class SimpleForm(forms.Form):
    birth_year = forms.DateField(widget=SelectDateWidget(years=B
IRTH_YEAR_CHOICES))
    favorite_colors = forms.MultipleChoiceField(required=False,
        widget=forms.CheckboxSelectMultiple, choices=FAVORITE_CO
LORS_CHOICES)
```

可用的Widget 以及它们接收的参数，参见*内建的Widget*。

继承自**Select** 的Widget

继承自 `Select` 的Widget 负责处理HTML 选项。它们呈现给用户一个可以选择的选项列表。不同的Widget 以不同的方式呈现选项； `Select` 使用HTML 的列表形式 `<select>` ，而 `RadioSelect` 使用单选按钮。

`ChoiceField` 字段默认使用 `Select` 。Widget 上显示的选项来自 `ChoiceField` ，对 `ChoiceField.choices` 的改变将更新 `Select.choices` 。例如：

```
>>> from django import forms
>>> CHOICES = (('1', 'First',), ('2', 'Second',))
>>> choice_field = forms.ChoiceField(widget=forms.RadioSelect, c
hoices=CHOICES)
>>> choice_field.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices = ()
>>> choice_field.choices = (('1', 'First and only',),)
>>> choice_field.widget.choices
[('1', 'First and only')]
```

提供 `choices` 属性的Widget 也可以用于不是基于选项的字段 ， 例如 `CharField` —— 当选项与模型有关而不只是Widget 时，建议使用基于 `ChoiceField` 的字段。

自定义Widget 的实例

当Django 渲染Widget 成HTML 时，它只渲染最少的标记 —— Django 不会添加class 的名称和特定于Widget 的其它属性。这表示，网页上所有 `TextInput` 的外观是一样的。

有两种自定义Widget 的方式：基于每个*Widget 实例*和基于每个*Widget 类*。

设置Widget 实例的样式

如果你想让某个Widget 实例与其它Widget 看上去不一样，你需要在Widget 对象实例化并赋值给一个表单字段时指定额外的属性（以及可能需要在你的CSS 文件中添加一些规则）。

例如下面这个简单的表单：

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()
```

这个表单包含三个默认的 `TextInput` Widget，以默认的方式渲染 —— 没有CSS 类、没有额外的属性。这表示每个Widget 的输入框将渲染得一模一样：

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" /></td></tr
>
<tr><th>Url:</th><td><input type="url" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></t
d></tr>
```

在真正得网页中，你可能不想让每个Widget 看上去都一样。你可能想要给comment 一个更大的输入元素，你可能想让'name' Widget 具有一些特殊的CSS 类。可以指定'type' 属性来利用新式的HTML5 输入类型。在创建Widget 时使用 `Widget.attrs` 参数可以实现：

```
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class'
: 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'siz
e': '40'}))
```

Django 将在渲染的输出中包含额外的属性：

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" class="spec
ial"/></td></tr>
<tr><th>Url:</th><td><input type="url" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size=
"40"/></td></tr>
```

你还可以使用 `attrs` 设置HTML `id` 。参见 `BoundField.id_for_label` 示例。

设置Widget 类的样式

可以添加（ `css` 和 `javascript` ）给Widget，以及深度定制它们的外观和行为。

概况来讲，你需要子类化Widget 并定义一个*“Media”* 内联类 或 创建一个*“media”* 属性。

这些方法涉及到Python 高级编程，详细细节在表单*Assets* 主题中讲述。

Widget 的基类

`Widget` 和 `MultiWidget` 是所有内建*Widget* 的基类，并可用于自定义Widget 的基类。

class `Widget` (*attrs=None*)

这是个抽象类，它不可以渲染，但是提供基本的属性 `attrs` 。你可以在自定义的Widget 中实现或覆盖 `render()` 方法。

`attrs`

包含渲染后的Widget 将要设置的HTML 属性。

```
>>> from django import forms
>>> name = forms.TextInput(attrs={'size': 10, 'title': 'Your nam
e',})
>>> name.render('name', 'A name')
'<input title="Your name" type="text" name="name" value="A name"
 size="10" />'
```

Changed in Django 1.8:

如果你给一个属性赋值 `True` 或 `False` ，它将渲染成一个HTML5 风格的布尔属性：

```
>>> name = forms.TextInput(attrs={'required': True})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" required />'
>>>
>>> name = forms.TextInput(attrs={'required': False})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" />'
```

`render` (*name*, *value*, *attrs=None*)

返回Widget 的HTML，为一个Unicode 字符串。子类必须实现这个方法，否则将引发 `NotImplementedError` 。

它不会确保给出的'value' 是一个合法的输入，因此子类的实现应该防卫式地编程。

`value_from_datadict` (*data*, *files*, *name*)

根据一个字典和该Widget 的名称，返回该Widget 的值。 `files` may contain data coming from `request.` FILES. 如果没有提供value，则返回 `None` 。 在处理表单数据的过程中， `value_from_datadict` 可能调用多次，所以如果你自定义并添加额外的耗时处理时，你应该自己实现一些缓存机制。

class `MultiWidget` (*widgets*, *attrs=None*)

由多个Widget 组合而成的Widget。 `MultiWidget` 始终与 `MultiValueField` 联合使用。

`MultiWidget` 具有一个必选参数：

`widgets`

一个包含需要的Widget 的可迭代对象。

以及一个必需的方法：

`decompress` (*value*)

这个方法接受来自字段的一个"压缩"的值，并返回"解压"的值的一个列表。可以假设输入的值是合法的，但不一定是非空的。

子类必须实现 这个方法，而且因为值可能为空，实现必须要防卫这点。

"解压"的基本原理是需要"分离"组合的表单字段的值为每个Widget 的值。

有个例子是， `SplitDateTimeWidget` 将 `datetime` 值分离成两个独立的值分别表示日期和时间：

```
from django.forms import MultiWidget

class SplitDateTimeWidget(MultiWidget):

    # ...

    def decompress(self, value):
        if value:
            return [value.date(), value.time().replace(microseco
nd=0)]
        return [None, None]
```

小贴士

注意， `MultiValueField` 有一个 `compress()` 方法用于相反的工作 —— 将所有字段的值组合成一个值。

其它可能需要覆盖的方法：

`render` (*name*, *value*, *attrs=None*)

这个方法中的 `value` 参数的处理方式与 `Widget` 子类不同，因为需要弄清楚如何为了在不同widget中展示分割单一值。

渲染中使用的 `value` 参数可以是二者之一：

- 一个 `列表` 。
- 一个单一值（比如字符串），它是 `列表` 的“压缩”表现形式。

如果 `value` 是个列表， `render()` 的输出会是一系列渲染后的子widget。如果 `value` 不是一个列表，首先会通过 `decompress()` 方法来预处理，创建列表，之后再渲染。

`render()` 方法执行HTML渲染时，列表中的每个值都使用相应的widget来渲染 -- 第一个值在第一个widget中渲染，第二个值在第二个widget中渲染，以此类推。

不像单一值的widget， `render()` 方法并不需要在子类中实现。

`format_output` (*rendered_widgets*)

接受选然后的widget（以字符串形式）的一个列表，返回表示全部HTML的Unicode字符串。

这个钩子允许你以任何你想要的方式，格式化widget的HTML设计。

下面示例中的Widget 继承 `MultiWidget` 以在不同的选择框中显示年、月、日。这个Widget 主要想用于 `DateField` 而不是 `MultiValueField` ，所以我们实现了 `value_from_datadict()` ：

```
from datetime import date
from django.forms import widgets

class DateSelectorWidget(widgets.MultiWidget):
    def __init__(self, attrs=None):
        # create choices for days, months, years
        # example below, the rest snipped for brevity.
        years = [(year, year) for year in (2011, 2012, 2013)]
        _widgets = (
            widgets.Select(attrs=attrs, choices=days),
            widgets.Select(attrs=attrs, choices=months),
            widgets.Select(attrs=attrs, choices=years),
        )
        super(DateSelectorWidget, self).__init__(_widgets, attrs
)

    def decompress(self, value):
        if value:
            return [value.day, value.month, value.year]
        return [None, None, None]

    def format_output(self, rendered_widgets):
        return ''.join(rendered_widgets)

    def value_from_datadict(self, data, files, name):
        datelist = [
            widget.value_from_datadict(data, files, name + '_%s'
 % i)
            for i, widget in enumerate(self.widgets)]
        try:
            D = date(day=int(datelist[0]), month=int(datelist[1]
),
                    year=int(datelist[2]))
        except ValueError:
            return ''
        else:
            return str(D)
```

构造器在一个元组中创建了多个 `Select` widget。 `超` 类使用这个元组来启动widget。

`format_output()` 方法相当于在这里没有干什么新的事情（实际上，它和 `MultiWidget` 中默认实现的东西相同），但是这个想法是，你可以以自己的方式在widget之间添加自定义的HTML。

必需的 `decompress()` 方法将 `datetime.date` 值拆成年、月和日的值，对应每个widget。注意这个方法如何处理 `value` 为 `None` 的情况。

`value_from_datadict()` 的默认实现会返回一个列表，对应每一个 `Widget` 。当和 `MultiValueField` 一起使用 `MultiWidget` 的时候，这样会非常合理，但是由于我们想要和拥有单一值得 `DateField` 一起使用这个widget，我们必须覆写这一方法，将所有子widget的数据组装成 `datetime.date` 。这个方法从 `POST` 字典中获取数据，并且构造和验证日期。如果日期有效，会返回它的字符串，否则会返回一个空字符串，它会使 `form.is_valid` 返回 `False` 。

内建的Widget

Django 提供所有基本的HTML Widget，并在 `django.forms.widgets` 模块中提供一些常见的Widget 组，包括文本的输入、各种选择框、文件上传和多值输入。

处理文本输入的Widget

这些Widget 使用HTML 元素 `input` 和 `textarea` 。

TextInput

class `TextInput`

文本输入： `<input type="text" ...>`

NumberInput

class `NumberInput`

文本输入： `<input type="number" ...>`

注意，不是所有浏览器的 `number` 输入类型都支持输入本地化的数字。Django 将字段的 `localize` 属性设置为 `True` 以避免字段使用它们。

EmailInput

class `EmailInput`

文本输入： `<input type="email" ...>`

URLInput

class `URLInput`

文本输入： `<input type="url" ...>`

PasswordInput

class `PasswordInput`

密码输入： `<input type='password' ...>`

接收一个可选的参数：

`render_value`

决定在验证错误后重新显示表单时，Widget 是否填充（默认为 `False`）。

HiddenInput

class `HiddenInput`

隐藏的输入： `<input type='hidden' ...>`

注意，还有一个 `MultipleHiddenInput` Widget，它封装一组隐藏的输入元素。

DateInput

class `DateInput`

日期以普通的文本框输入： `<input type='text' ...>`

接收的参数与 `TextInput` 相同，但是带有一些可选的参数：

`format`

字段的初始值应该显示的格式。

如果没有提供 `format` 参数，默认的格式为参考本地化格式在 `DATE_INPUT_FORMATS` 中找到的第一个格式。

DateTimeInput

class `DateTimeInput`

日期/时间以普通的文本框输入： `<input type='text' ...>`

接收的参数与 `TextInput` 相同，但是带有一些可选的参数：

`format`

字段的初始值应该显示的格式。

如果没有提供 `format` 参数，默认的格式为参考本地化格式在 `DATETIME_INPUT_FORMATS` 中找到的第一个格式。

TimeInput

class `TimeInput`

时间以普通的文本框输入： `<input type='text' ...>`

接收的参数与 `TextInput` 相同，但是带有一些可选的参数：

`format`

字段的初始值应该显示的格式。

如果没有提供 `format` 参数，默认的格式为参考本地化格式在 `TIME_INPUT_FORMATS` 中找到的第一个格式。

Textarea

class `Textarea`

文本区域： `<textarea>...</textarea>`

选择和复选框Widget

CheckboxInput

class `CheckboxInput`

复选框： `<input type='checkbox' ...>`

接受一个可选的参数：

`check_test`

一个可调用的对象，接收 `CheckboxInput` 的值并如果复选框应该勾上返回 `True` 。

Select

class `Select`

Select widget： `<select><option ...>...</select>`

`choices`

当表单字段没有 `choices` 属性时，该属性是随意的。如果字段有choice 属性，当 `字段` 的该属性更新时，它将覆盖你在这里的任何设置。

NullBooleanSelect

class `NullBooleanSelect`

Select Widget，选项为'Unknown'、'Yes' 和'No'。

SelectMultiple

class `SelectMultiple`

类似 `Select` ，但是允许多个选择： `<select multiple='multiple'>...</select>`

RadioSelect

class `RadioSelect`

类似 `Select` ，但是渲染成 `<li>` 标签中的一个单选按钮列表：

```
<ul>
  <li><input type='radio' name='...'></li>
  ...
</ul>
```

你可以迭代模板中的单选按钮来更细致地控制生成的HTML。假设表单 `myform` 具有一个字段 `beatles` ，它使用 `RadioSelect` 作为Widget：

```
{% for radio in myform.beatles %}
<div class="myradio">
    {{ radio }}
</div>
{% endfor %}
```

它将生成以下HTML：

```
<div class="myradio">
    <label for="id_beatles_0"><input id="id_beatles_0" name="bea
tles" type="radio" value="john" /> John</label>
</div>
<div class="myradio">
    <label for="id_beatles_1"><input id="id_beatles_1" name="bea
tles" type="radio" value="paul" /> Paul</label>
</div>
<div class="myradio">
    <label for="id_beatles_2"><input id="id_beatles_2" name="bea
tles" type="radio" value="george" /> George</label>
</div>
<div class="myradio">
    <label for="id_beatles_3"><input id="id_beatles_3" name="bea
tles" type="radio" value="ringo" /> Ringo</label>
</div>
```

这包括 `<label>` 标签。你可以使用单选按钮的 `tag` 、 `choice_label` 和 `id_for_label` 属性进行更细的控制。例如，这个模板：

```
{% for radio in myform.beatles %}
    <label for="{{ radio.id_for_label }}">
        {{ radio.choice_label }}
        <span class="radio">{{ radio.tag }}</span>
    </label>
{% endfor %}
```

... 将生成下面的HTML：

```
<label for="id_beatles_0">
    John
    <span class="radio"><input id="id_beatles_0" name="beatles"
type="radio" value="john" /></span>
</label>

<label for="id_beatles_1">
    Paul
    <span class="radio"><input id="id_beatles_1" name="beatles"
type="radio" value="paul" /></span>
</label>

<label for="id_beatles_2">
    George
    <span class="radio"><input id="id_beatles_2" name="beatles"
type="radio" value="george" /></span>
</label>

<label for="id_beatles_3">
    Ringo
    <span class="radio"><input id="id_beatles_3" name="beatles"
type="radio" value="ringo" /></span>
</label>
```

如果你不迭代单选按钮 —— 例如，你的模板只是简单地包含 `{{ myform.beatles }}` —— 它们将以 `<ul>` 中的 `<li>` 标签输出，就像上面一样。

外层的 `<ul>` 将带有定义在Widget 上的 `id` 属性。

Changed in Django 1.7:

当迭代单选按钮时， `label` 和 `input` 标签分别包含 `for` 和 `id` 属性。每个单项按钮具有一个 `id_for_label` 属性来输出元素的ID。

CheckboxSelectMultiple

class `CheckboxSelectMultiple`

类似 `SelectMultiple` ，但是渲染成一个复选框列表：

```
<ul>
  <li><input type='checkbox' name='...' ></li>
  ...
</ul>
```

外层的 `<ul>` 具有定义在Widget 上的 `id` 属性。

类似 `RadioSelect` ，你可以迭代列表的每个复选框。更多细节参见 `RadioSelect` 的文档。

Changed in Django 1.7:

当迭代单选按钮时， `label` 和 `input` 标签分别包含 `for` 和 `id` 属性。 每个单项按钮具有一个 `id_for_label` 属性来输出元素的ID。

文件上传Widget

FileInput

class `FileInput`

文件上传输入： `<input type='file' ...>`

ClearableFileInput

class `ClearableFileInput`

文件上传输入： `<input type='file' ...>` ，带有一个额外的复选框，如果该字段不是必选的且有初始的数据，可以清除字段的值。

复合Widget

MultipleHiddenInput

class `MultipleHiddenInput`

多个 `<input type='hidden' ...>` Widget。

一个处理多个隐藏的Widget 的Widget，用于值为一个列表的字段。

`choices`

当表单字段没有 `choices` 属性时，这个属性是可选的。如果字段有choice 属性，当 `字段` 的该属性更新时，它将覆盖你在这里的任何设置。

SplitDateTimeWidget

class `SplitDateTimeWidget`

封装（使用 `MultiWidget` ）两个Widget： `DateInput` 用于日期， `TimeInput` 用于时间。

`SplitDateTimeWidget` 有两个可选的属性：

`date_format`

类似 `DateInput.format`

`time_format`

类似 `TimeInput.format`

SplitHiddenDateTimeWidget

class `SplitHiddenDateTimeWidget`

类似 `SplitDateTimeWidget` ，但是日期和时间都使用 `HiddenInput` 。

SelectDateWidget

class `SelectDateWidget` [source]

封装三个 `Select` Widget：分别用于年、月、日。注意，这个Widget 与标准的Widget 位于不同的文件中。

接收一个可选的参数：

`years`

一个可选的列表/元组，用于"年"选择框。默认为包含当前年份和未来9年的一个列表。

`months`

New in Django 1.7.

一个可选的字典，用于"月"选择框。

字典的键对应于月份的数字（从1开始），值为显示出来的月份：

```
MONTHS = {
    1:_('jan'), 2:_('feb'), 3:_('mar'), 4:_('apr'),
    5:_('may'), 6:_('jun'), 7:_('jul'), 8:_('aug'),
    9:_('sep'), 10:_('oct'), 11:_('nov'), 12:_('dec')
}
```

`empty_label`

New in Django 1.8.

如果 `DateField` 不是必选的， `SelectDateWidget` 将有一个空的选项位于选项的顶部（默认为 `---` ）。你可以通过 `empty_label` 属性修改这个文本。 `empty_label` 可以是一个 `字符串` 、 `列表` 或 `元组` 。当使用字符串时，所有的选择框都带有这个空选项。如果 `empty_label` 为具有3个字符串元素的 `列表` 或 `元组` ，每个选择框将具有它们自定义的空选项。空选项应该按这个顺序 `('year_label', 'month_label', 'day_label')` 。

```
# A custom empty label with string
field1 = forms.DateField(widget=SelectDateWidget(empty_label="No
thing"))

# A custom empty label with tuple
field1 = forms.DateField(widget=SelectDateWidget(
    empty_label=("Choose Year", "Choose Month", "Choose Day"))
```

> 译者：Django 文档协作翻译小组，原文：Built-in widgets。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

高级

从模型创建表单

ModelForm

class `ModelForm`

如果你正在构建一个数据库驱动的应用，那么你应该会有与Django 的模型紧密映射的表单。举个例子，你也许会有个 `BlogComment` 模型，并且你还想创建一个表单让大家提交评论到这个模型中。 在这种情况下，在表单中定义字段将是冗余的，因为你已经在模型中定义了字段。

基于这个原因，Django 提供一个辅助类来让你可以从Django 的模型创建 `表单` 。

例如：

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'report
er']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

字段类型

生成的 `表单` 类中将具有和指定的模型字段对应的表单字段，顺序为 `fields` 属性中指定的顺序。

每个模型字段有一个对应的默认表单字段。比如，模型中的 `CharField` 表现成表单中的 `CharField` 。模型中的 `ManyToManyField` 字段会表现成 `MultipleChoiceField` 字段。下面是一个完整的列表：

Model field	Form field
`AutoField`	Not represented in the form
	`IntegerField` with `min_value` set to

BigIntegerField	-9223372036854775808 and max_value set to 9223372036854775807.
BooleanField	BooleanField
CharField	CharField with max_length set to the model field's max_length
CommaSeparatedIntegerField	CharField
DateField	DateField
DateTimeField	DateTimeField
DecimalField	DecimalField
EmailField	EmailField
FileField	FileField
FilePathField	FilePathField
FloatField	FloatField
ForeignKey	ModelChoiceField (see below)
ImageField	ImageField
IntegerField	IntegerField
IPAddressField	IPAddressField
GenericIPAddressField	GenericIPAddressField
ManyToManyField	ModelMultipleChoiceField (see below)
NullBooleanField	NullBooleanField
PositiveIntegerField	IntegerField
PositiveSmallIntegerField	IntegerField
SlugField	SlugField
SmallIntegerField	IntegerField
TextField	CharField with widget=forms.Textarea
TimeField	TimeField
URLField	URLField

可能如你所料， ForeignKey 和 ManyToManyField 字段类型属于特殊情况：

- ForeignKey 表示成 django.forms.ModelChoiceField ，它是一个 ChoiceField ，其选项是模型的 查询集 。

- `ManyToManyField` 表示成 `django.forms.ModelMultipleChoiceField` ，它是一个 `MultipleChoiceField` ，其选项是模型的 `查询集` 。

此外，生成的每个表单字段都有以下属性集：

- 如果模型字段设置 `blank=True` ，那么表单字段的 `required` 设置为 `False` 。否则， `required=True` 。
- 表单字段的 `label` 设置为模型字段的 `verbose_name` ，并将第一个字母大写。
- 表单字段的 `help_text` 设置为模型字段的 `help_text` 。
- 如果模型字段设置了 `choices` ，那么表单字段的 `Widget` 将设置成 `Select` ，其选项来自模型字段的 `choices` 。选项通常会包含空选项，并且会默认选择。如果字段是必选的，它会强制用户选择一个选项。如果模型字段的 `blank=False` 且具有一个显示的 `default` 值，将不会包含空选项（初始将选择 `default` 值）。

最后，请注意你可以为给定的模型字段重新指定表单字段。参见下文覆盖默认的字段。

一个完整的例子

考虑下面的模型：

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = (
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
)

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES
)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['name', 'authors']
```

上面 `ModelForm` 的子类大体等同于（唯一的不同是 `save()` 方法，我们将稍后讨论）：

```
from django import forms

class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(max_length=3,
                widget=forms.Select(choices=TITLE_CHOICES))
    birth_date = forms.DateField(required=False)

class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.obj
ects.all())
```

`模型表单` 的验证

验证 `模型表单` 主要有两步：

1. 验证表单
2. 验证模型实例

与普通的表单验证类型类似，模型表单的验证在调用 `is_valid()` 或访问 `errors` 属性时隐式调用，或者通过 `full_clean()` 显式调用，尽管在实际应用中你将很少使用后一种方法。

`模型` 的验证（ `Model.full_clean()` ）在表单验证这一步的内部触发，紧跟在表单的 `clean()` 方法调用之后。

警告

Clean 过程会以各种方式修改传递给 `模型表单` 构造函数的模型实例。例如，模型的日期字段将转换成日期对象。验证失败可能导致模型实例处于不一致的状态，所以不建议重新使用它。

重写clean() 方法

可以重写模型表单的 `clean()` 来提供额外的验证，方法和普通的表单一样。

模型表单实例包含一个 `instance` 属性，表示与它绑定的模型实例。

警告

`ModelForm.clean()` 方法设置一个标识符， 使得模型验证 这一步验证标记为 `unique` 、 `unique_together` 或 `unique_for_date|month|year` 的模型字段的唯一性。

如果你需要覆盖 `clean()` 方法并维持这个验证行为，你必须调用父类的 `clean()` 方法。

与模型验证的交互

作为验证过程的一部分， `模型表单` 将调用与表单字段对应的每个模型字段的 `clean()` 方法。如果你已经排除某些模型字段，这些字段不会运行验证。关于字段clean 和验证是如何工作的，参见表单字段的文档。

模型的 `clean()` 方法在任何唯一性检查之前调用。关于模型 `clean()` 钩子的更多信息，参见验证对象 。

模型 `error_messages` 的注意事项

`表单字段`级别或表单级别的错误信息永远比 `模型字段` 级别的错误信息优先。

`模型字段` 的错误信息只用于模型验证步骤引发 `ValidationError` 的时候，且不会有对应的表单级别的错误信息。

New in Django 1.7.

你可以根据模型验证引发的 `NON_FIELD_ERRORS` 覆盖错误信息，方法是添加 `NON_FIELD_ERRORS` 键到 `模型表单` 内联 `Meta` 类的 `error_messages` 字典：

```
from django.forms import ModelForm
from django.core.exceptions import NON_FIELD_ERRORS

class ArticleForm(ModelForm):
    class Meta:
        error_messages = {
            NON_FIELD_ERRORS: {
                'unique_together': "%(model_name)s's %(field_lab
els)s are not unique.",
            }
        }
```

`save()` 方法

每个 `模型表单` 还具有一个 `save()` 方法。这个方法根据表单绑定的数据创建并保存数据库对象。 `模型表单` 的子类可以用关键字参数 `instance` 接收一个已经存在的模型实例；如果提供， `save()` 将更新这个实例。如果没有提供， `save()` 将创建模型的一个新实例：

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

注意，如果表单没有验证， `save()` 调用将通过检查 `form.errors` 来进行验证。如果表单中的数据不合法，将引发 `ValueError` —— 例如，如果 `form.errors` 为 `True` 。

`save()` 接受一个可选的 `commit` 关键字参数，其值为 `True` 或 `False` 。如果 `save()` 时 `commit=False` ，那么它将返回一个还没有保存到数据库的对象。这种情况下，你需要调用返回的模型实例的 `save()` 。 如果你想在保存之前自定义一些处理，或者你想使用特定的模型保存选项，可以这样使用。 `commit` 默认为 `True` 。

使用 `commit=False` 的另外一个副作用是在模型具有多对多关系的时候。如果模型具有多对多关系而且当你保存表单时指定 `commit=False` ，Django 不会立即为多对多关系保存表单数据。这是因为只有实例在数据库中存在时才可以保存实例的多对多数据。

为了解决这个问题，每当你使用 `commit=False` 保存表单时，Django 将添加一个 `save_m2m()` 方法到你的 `模型表单` 子类。在你手工保存有表单生成的实例之后，你可以调用 `save_m2m()` 来保存多对多的表单数据。例如：

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = 'some_value'

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

`save_m2m()` 只在你使用 `save(commit=False)` 时才需要。当你直接使用 `save()` ，所有的数据 —— 包括多对多数据 —— 都将保存而不需要任何额外的方法调用。例如：

```
# Create a form instance with POST data.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do
 anything else.
>>> new_author = f.save()
```

除了 `save()` 和 `save_m2m()` 方法之外， `模型表单` 与其它 `表单` 的工作方式完全一样。例如， `is_valid()` 用于检查合法性， `is_multipart()` 方法用于决定表单是否需要multipart 的文件上传（以及这之后 `request.FILES` 是否必须必须传递给表单）等等。更多信息，参见绑定上传的文件到表单。

选择用到的字段

强烈建议你使用 `fields` 属性显式设置所有将要在表单中编辑的字段。如果不这样做，当表单不小心允许用户设置某些特定的字段，特别是有的字段添加到模型中的时候，将很容易导致安全问题。这些问题可能在网页上根本看不出来，它与表单的渲染方式有关。

另外一种方式是自动包含所有的字段，或者排除某些字段。这种基本方式的安全性要差很多，而且已经导致大型的网站受到严重的利用（例如 GitHub）。

然而，有两种简单的方法保证你不会出现这些安全问题：

1. 设置 `fields` 属性为特殊的值 `'__all__'` 以表示需要使用模型的所有字段。例如：

   ```
   from django.forms import ModelForm

   class AuthorForm(ModelForm):
       class Meta:
           model = Author
           fields = '__all__'
   ```

2. 设置 `ModelForm` 内联的 `Meta` 类的 `exclude` 属性为一个要从表单中排除的字段的列表。

   例如：

   ```
   class PartialAuthorForm(ModelForm):
       class Meta:
           model = Author
           exclude = ['title']
   ```

   因为 `Author` 模型有3个字段 `name` 、 `title` 和 `birth_date` ，上面的例子会让 `name` 和 `birth_date` 出现在表单中。

如果使用上面两种方法，表单中字段出现的顺序将和字段在模型中定义的顺序一致，其中 `ManyToManyField` 出现在最后。

另外，Django 还将使用以下规则：如果设置模型字段的 `editable=False` ，那么使用 `ModelForm` 从该模型创建的任何表单都不会包含该字段。

Changed in Django 1.8:

在旧的版本中，同时省略 `fields` 和 `exclude` 字段将导致模型的所有字段出现在表单中。现在这样做将引发一个 `ImproperlyConfigured` 异常。

注

不会被上述逻辑包含进表单中的字段将不会被表单的 `save()` 方法保存。另外，如果你手工添加排除的字段到表单中，它们也不会从模型实例初始化。

Django 将阻止保存不完全的模型，所以如果模型不允许缺失的字段为空且没有提供默认值，带有缺失字段的 `ModelForm` 的 `save()` 将失败。为了避免这种失败，实例化模型时必须带有缺失的字段的初始值：

```
author = Author(title='Mr')
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

还有一种方法，你可以使用 `save(commit=False)` 并手工设置额外需要的字段：

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = 'Mr'
author.save()
```

关于使用 `save(commit=False)` 的更多细节参见保存表单一节。

重写（覆盖）默认的字段

上文字段类型表中默认的字段类型只是合理的默认值。如果你的模型中有一个 `DateField` ，你可能想在表单中也将它表示成 `DateField` 。但是 `ModelForm` 还提供更多的灵活性，让你可以改变给定的模型字段对应的表单字段的类型和Widget。

使用内部类 `Meta` 的 `widgets` 属性可以指定一个字段的自定义Widget。它是映射字段名到Widget 类或实例的一个字典。

例如， `Author` 的 `name` 属性为 `CharField` ，如果你希望它表示成一个 `<textarea>` 而不是默认的 `<input type="text">` ，你可以覆盖字段默认的Widget：

```
from django.forms import ModelForm, Textarea
from myapp.models import Author

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        widgets = {
            'name': Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

不管是Widget 实例（ `Textarea(...)` ）还是Widget 类（ `Textarea` ）， `widgets` 字典都可以接收。

类似地，如果你希望进一步自定义字段，你可以指定内部类 `Meta` 的 `labels` 、 `help_texts` 和 `error_messages` 。

例如，如果你希望自定义 `name` 字段所有面向用户的字符串：

```
from django.utils.translation import ugettext_lazy as _

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        labels = {
            'name': _('Writer'),
        }
        help_texts = {
            'name': _('Some useful help text.'),
        }
        error_messages = {
            'name': {
                'max_length': _("This writer's name is too long.
"),
            },
        }
```

最后，如果你希望完全控制字段 —— 包括它的类型、验证器等等，你可以像在普通的 `表单` 那样显式指定字段。

例如，如果你想为 `slug` 字段使用 `MySlugFormField` ，可以像下面这样：

```
from django.forms import ModelForm
from myapp.models import Article

class ArticleForm(ModelForm):
    slug = MySlugFormField()

    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter',
 'slug']
```

如果想要指定字段的验证器，可以显式定义字段并设置它的 `validators` 参数：

```
from django.forms import ModelForm, CharField
from myapp.models import Article

class ArticleForm(ModelForm):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter',
 'slug']
```

注

当你像这样显式实例化表单字段时，需要理解 `ModelForm` 和普通的 `Form` 的关系是怎样的。

`ModelForm` 就是可以自动生产相应字段的 `Form` .自动生成哪些字段取决于 `Meta` 类的fields属性和在该ModelForm中显示声明的字段。 `ModelForm` 基本上只 生成表单中没有的字段，换句话讲就是没有显式定义的字段。

显式定义的字段会保持原样，所以 `Meta` 属性中任何自定义的属性例如 `widgets` 、 `labels` 、 `help_texts` 或 `error_messages` 都将忽略；它们只适用于自动生成的字段。

类似地，显式定义的字段不会从对应的模型中获取属性，例如 `max_length` 或 `required` 。 如果你希望保持模型中指定的行为，你必须设置在声明表单字段时显式设置相关的参数。

例如，如果 `Article` 模型像下面这样：

```
class Article(models.Model):
    headline = models.CharField(max_length=200, null=True, blank
=True,
                                help_text="Use puns liberally")
    content = models.TextField()
```

而你想为 `headline` 做一些自定义的验证，在保持 `blank` 和 `help_text` 值的同时，你必须这样定义 `ArticleForm` ：

```
class ArticleForm(ModelForm):
    headline = MyFormField(max_length=200, required=False,
                           help_text="Use puns liberally")

    class Meta:
        model = Article
        fields = ['headline', 'content']
```

你必须保证表单字段的类型可以用于对应的模型字段。如果它们不兼容，因为不会有显示的转换你将会得到一个 `ValueError` 。

关于字段和它们的参数，参见表单字段的文档。

启用字段的本地化功能

默认情况下， `ModelForm` 中的字段不会本地化它们的数据。你可以使用 `Meta` 类的 `localized_fields` 属性来启用字段的本地化功能。

```
>>> from django.forms import ModelForm
>>> from myapp.models import Author
>>> class AuthorForm(ModelForm):
...     class Meta:
...         model = Author
...         localized_fields = ('birth_date',)
```

如果 `localized_fields` 设置为 `'__all__'` 这个特殊的值，所有的字段都将本地化。

表单继承

在基本的表单里，你可以通过继承 `ModelForms` 来扩展和重用他们。当你的form是通过models生成的，而且需要在父类的基础上声明额外的field和method，这种继承是方便的。例如，使用以前的 `ArticleForm` 类：

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self):
...         ...
```

以上创建了一个与 `ArticleForm` 非常类似的form，除了一些额外的验证和 `pub_date` 的cleaning

你也可以在子类中继承父类的内部类 `Meta` 来重写它的属性列表，比如 `Meta.fields` 或者 `Meta.excludes` ：

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ('body',)
```

上例从父类 `EnhancedArticleForm` 继承后增加了额外的方法，并修改了 `ArticleForm.Meta` 排除了一个字段

当然，有一些注意事项

- 应用正常的Python名称解析规则。如果你有多个基类声明一个 `Meta` 内部类，只会使用第一个。这意味着孩子的 `Meta` （如果存在），否则第一个父母的 `Meta` 等。

Changed in Django 1.7.

- 它可以同时继承 `Form` 和 `ModelForm` ，但是，必须确保 `ModelForm` 首先出现在MRO中。这是因为这些类依赖于不同的元类，而一个类只能有一个元类。

New in Django 1.7.

- 可以通过在子类上将名称设置为 `None` ，声明性地删除从父类继承的 `Field` 。

  您只能使用此技术选择退出由父类声明定义的字段；它不会阻止 `ModelForm` 元类生成默认字段。要退出默认字段，请参阅*Selecting the fields to use*。

提供初始值

作为一个有参数的表单, 在实例化一个表单时可以通过指定 `initial` 字段来指定表单中数据的初始值. 这种方式指定的初始值将会同时替换掉表单中的字段和值. 例如:

```
>>> article = Article.objects.get(pk=1)
>>> article.headline
'My headline'
>>> form = ArticleForm(initial={'headline': 'Initial headline'},
 instance=article)
>>> form['headline'].value()
'Initial headline'
```

模型表单的**factory**函数

你可以用单独的函数 `modelform_factory()` 来代替使用类定义来从模型直接创建表单。这在不需要很多自定义的情况下应该是更方便的。

```
>>> from django.forms.models import modelform_factory
>>> from myapp.models import Book
>>> BookForm = modelform_factory(Book, fields=("author", "title"
))
```

这个函数还能对已有的表单类做简单的修改，比如，对给出的字段指定 widgets ：

```
>>> from django.forms import Textarea
>>> Form = modelform_factory(Book, form=BookForm,
...                          widgets={"title": Textarea()})
```

表单包含的字段可以用 `fields` 或 `exclude` 关键字参数说明，或者用 `ModelForm` 内部 `Meta` 类的相应属性说明。请看 `ModelForm` 文档： 选择使用的字段。

... 或者为指定的字段启用本地化功能。

```
>>> Form = modelform_factory(Author, form=AuthorForm, localized_
fields=("birth_date",))
```

模型表单集

class `models.``BaseModelFormSet`

与普通表单集一样, 它是Django提供的几个有力的表单集类来简化模型操作。. 让我们继续使用上面的 `Author` 模型：

```
>>> from django.forms.models import modelformset_factory
>>> from myapp.models import Author
>>> AuthorFormSet = modelformset_factory(Author, fields=('name',
 'title'))
```

使用 `fields` 限定表单集仅可以使用给出的字段，或者使用排除法，指定哪些字段被不被使用。

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=('birth
_date',))
```

Changed in Django 1.8:

在旧版本中，同时省略 `fields` 和 `exclude` 的结果是表单集使用模型的所有字段。现在这么做将引发 `ImproperlyConfigured` 异常。

下面将创建一个与 `Author` 模型数据相关联的功能强大的表单集，与普通表单集运行一样：

```
>>> formset = AuthorFormSet()
>>> print(formset)
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS" /><input type="hidden" name="form-INITIAL_FORMS" value="0" id="id_form-INITIAL_FORMS" /><input type="hidden" name="form-MAX_NUM_FORMS" id="id_form-MAX_NUM_FORMS" />
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name="form-0-name" maxlength="100" /></td></tr>
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-title" id="id_form-0-title">
<option value="" selected="selected">---------</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select><input type="hidden" name="form-0-id" id="id_form-0-id" /></td></tr>
```

注意

`modelformset_factory()` 使用 `formset_factory()` 生成表单集，这意味着模型表单集仅仅是扩展基本表单集，使其能处理模型的信息。

更改查询集

默认的, 如果你使用model生成formset，formset会使用一个包含模型全部对象的queryset(例如: `Author.objects.all()`). 你可以使用 `queryset` 参数重写这一行为：

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
```

或者，你可以创建子类设置 `self.queryset` in `__init__` :

```
from django.forms.models import BaseModelFormSet
from myapp.models import Author

class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
        self.queryset = Author.objects.filter(name__startswith='O')
```

然后，将 `BaseAuthorFormSet` 类传给modelformset_factory函数：

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title'), formset=BaseAuthorForm
Set)
```

如果想返回不包含任何已存在模型实例的表单集，可以指定一个空的查询集（QuerySet）：

```
>>> AuthorFormSet(queryset=Author.objects.none())
```

更改 `form`

默认情况下,当你使用 `modelformset_factory` 时, `modelform_factory()` 将会创建一个模型 通常这有助于指定一个自定义模型表单. 例如,你可以创建一个自定义验证的表单模型

```
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

    def clean_name(self):
        # custom validation for the name field
        ...
```

然后,把你的模型作为参数传递过去

```
AuthorFormSet = modelformset_factory(Author, form=AuthorForm)
```

并不总是需要自定义一个模型表单， `modelformset_factory` 函数有几个参数，可以传给 `modelform_factory` ，他们的说明如下：

指定要在 `widgets` 中使用的小部件

使用 `widgets` 参数，可以用字典值自定义 `ModelForm` 列出字段的widget类。这与 `widgets` 字典在 `ModelForm` 的内部 `Meta` 类作用式一样。

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title'),
...     widgets={'name': Textarea(attrs={'cols': 80, 'rows': 20}
)})
```

使用 `localized_fields` 为字段启用本地化

使用 `localized_fields` 参数，可以使表单中字段启用本地化。

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title', 'birth_date'),
...     localized_fields=('birth_date',))
```

如果 `localized_fields` 设置值为 `'__all__'` ，将本地化所有字段。

提供初始化数据

与普通表单集一样， `modelformset_factory()` 能返回初始化的模型表单集， `initial` 参数能为表单集的中表单指定初始数据 。但是，在模型表单集中，初始数据仅应用在增加的表单中，不会应用到已存在的模型实例。如果用户没有更改新增加表单中的初始数据，那他们也不会被校验和保存。

保存表单集中的对象

做为 `ModelForm` , 你可以保存数据到模型对象，以下就完成了表单集的 `save()` 方法:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

`save()` 方法返回已保存到数据库的实例。如果给定实例的数据在绑定数据中没有更改，那么实例将不会保存到数据库，并且不会包含在返回值中（在上面的示例中为 `instances` ）。

当窗体中缺少字段（例如因为它们已被排除）时，这些字段不会由 `save()` 方法设置。您可以在选择要使用的字段中找到有关此限制的更多信息，这也适用于常规 `ModelForms` 。

传递 `commit=False` 返回未保存的模型实例：

```
# don't save to the database
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # do something with instance
...     instance.save()
```

这使您能够在将数据保存到数据库之前将数据附加到实例。如果您的表单集包含 `ManyToManyField` ，您还需要调用 `formset.save_m2m()` ，以确保多对多关系正确保存。

调用 `save()` 之后，您的模型formset将有三个包含formset更改的新属性：

`models.BaseModelFormSet.``changed_objects`

`models.BaseModelFormSet.``deleted_objects`

`models.BaseModelFormSet.``new_objects`

操作表单对像的数量限制

与普通表单集一样，你可以用在 `modelformset_factory()` 中使用 `max_num` 和 `extra` 参数，来控制额外表单的显示数量。

`max_num` 不会限制已经存在的表单对像的显示:

```
>>> Author.objects.order_by('name')
[<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author:
 Walt Whitman>]

>>> AuthorFormSet = modelformset_factory(Author, fields=('name',
), max_num=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('na
me'))
>>> [x.name for x in formset.get_queryset()]
['Charles Baudelaire', 'Paul Verlaine', 'Walt Whitman']
```

如果 `max_num` 大于存在的关联对像的数量，表单集将添加 `extra` 个额外的空白表单，只要表单总数量不超过 `max_num` ：

```
>>> AuthorFormSet = modelformset_factory(Author, fields=('name',
), max_num=4, extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('na
me'))
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-name">Name:</label></th><td><input
 id="id_form-0-name" type="text" name="form-0-name" value="Charl
es Baudelaire" maxlength="100" /><input type="hidden" name="form
-0-id" value="1" id="id_form-0-id" /></td></tr>
<tr><th><label for="id_form-1-name">Name:</label></th><td><input
 id="id_form-1-name" type="text" name="form-1-name" value="Paul
Verlaine" maxlength="100" /><input type="hidden" name="form-1-id
" value="3" id="id_form-1-id" /></td></tr>
<tr><th><label for="id_form-2-name">Name:</label></th><td><input
 id="id_form-2-name" type="text" name="form-2-name" value="Walt
Whitman" maxlength="100" /><input type="hidden" name="form-2-id"
 value="2" id="id_form-2-id" /></td></tr>
<tr><th><label for="id_form-3-name">Name:</label></th><td><input
 id="id_form-3-name" type="text" name="form-3-name" maxlength="1
00" /><input type="hidden" name="form-3-id" id="id_form-3-id" />
</td></tr>
```

`max_num` 值为f `None` （缺省）设置一个较高的限制可显示1000个表单。实际上相当于没有限制。

在视图中使用模型表单集

模型表单集与表单集十分类似，假设我们想要提供一个表单集来编辑 `Author` 模型实例：

```
from django.forms.models import modelformset_factory
from django.shortcuts import render_to_response
from myapp.models import Author

def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=('name',
 'title'))
    if request.method == 'POST':
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = AuthorFormSet()
    return render_to_response("manage_authors.html", {
        "formset": formset,
    })
```

可以看到，模型表单集的视图逻辑与“正常”表单集的视图逻辑没有显着不同。唯一的区别是我们调用 `formset.save()` 将数据保存到数据库中。（如上所述，在*Saving objects in the formset*中的对象）。

在 `ModelFormSet` 上覆盖 `clean()`

与 `ModelForms` 一样，默认情况下， `ModelFormSet` 的 `clean()` 方法将验证formset中没有项目违反唯一约束（ `unique` ， `unique_together` 或 `unique_for_date|month|year` ）。如果要覆盖 `ModelFormSet` 上的 `clean()` 方法并维护此验证，则必须调用父类的 `clean` 方法：

```
from django.forms.models import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

另请注意，到达此步骤时，已为每个 `Form` 创建了各个模型实例。修改 `form.cleaned_data` 中的值不足以影响保存的值。如果您希望修改 `ModelFormSet.clean()` 中的值，则必须修改 `form.instance` ：

```
from django.forms.models import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()

        for form in self.forms:
            name = form.cleaned_data['name'].upper()
            form.cleaned_data['name'] = name
            # update the instance value.
            form.instance.name = name
```

使用自定义queryset

如前所述，您可以覆盖模型formset使用的默认查询集：

```
from django.forms.models import modelformset_factory
from django.shortcuts import render_to_response
from myapp.models import Author

def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=('name',
 'title'))
    if request.method == "POST":
        formset = AuthorFormSet(request.POST, request.FILES,
                                queryset=Author.objects.filter(n
ame__startswith='O'))
        if formset.is_valid():
            formset.save()
            # Do something.
    else:
        formset = AuthorFormSet(queryset=Author.objects.filter(n
ame__startswith='O'))
    return render_to_response("manage_authors.html", {
        "formset": formset,
    })
```

请注意，我们在此示例中的 `POST` 和 `GET` 中传递 `queryset` 参数。

在模板中使用表单集

在Django模板中有三种方式来渲染表单集。

第一种方式，你可以让表单集完成大部分的工作

```
<form method="post" action="">
    {{ formset }}
</form>
```

其次，你可以手动渲染formset，但让表单处理自己：

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>
```

当您自己手动呈现表单时，请确保呈现如上所示的管理表单。请参阅*management form documentation*。

第三，您可以手动呈现每个字段：

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }} {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

如果您选择使用此第三种方法，并且不对 `{% for %} t0> loop，你需要渲染主键字段。` 例如，如果您要渲染模型的 `name` 和 `age` 字段：

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form.id }}
        <ul>
            <li>{{ form.name }}</li>
            <li>{{ form.age }}</li>
        </ul>
    {% endfor %}
</form>
```

注意我们需要如何显式渲染 `{{ form.id }}` 。这确保了在 `POST` 情况下的模型形式集将正常工作。（此示例假设名为 `id` 的主键。如果您明确定义了自己的主键（不是 `id` ），请确保其呈现）。

内联formets

class `models.``BaseInlineFormSet`

内联formets是模型formets上的一个小的抽象层。这些简化了通过外键处理相关对象的情况。假设你有这两个模型：

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=100)
```

如果要创建允许您编辑属于特定作者的图书的表单集，您可以执行以下操作：

```
>>> from django.forms.models import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('t
itle',))
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)
```

注意

`inlineformset_factory()` 使用 `modelformset_factory()` 并标记为 `can_delete=True` 。

也可以看看

Manually rendered can_delete and can_order。

覆盖 `InlineFormSet` 上的方法

当覆盖 `InlineFormSet` 上的方法时，您应该子类化 `BaseInlineFormSet` ，而不是 `BaseModelFormSet` 。

例如，如果要覆盖 `clean()` ：

```
from django.forms.models import BaseInlineFormSet

class CustomInlineFormSet(BaseInlineFormSet):
    def clean(self):
        super(CustomInlineFormSet, self).clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

另请参见*Overriding clean() on a ModelFormSet*上的clean()。

然后，在创建内联表单集时，传递可选参数 `formset` ：

```
>>> from django.forms.models import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('t
itle',),
...     formset=CustomInlineFormSet)
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)
```

多个外键对同一个模型

如果您的模型在同一个模型中包含多个外键，则需要使用 `fk_name` 手动解决歧义。例如，考虑以下模型：

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(Friend, related_name='from_friends')
    to_friend = models.ForeignKey(Friend, related_name='friends')
    length_in_months = models.IntegerField()
```

要解决此问题，您可以使用 `fk_name` 到 `inlineformset_factory()` ：

```
>>> FriendshipFormSet = inlineformset_factory(Friend, Friendship, fk_name='from_friend',
...     fields=('to_friend', 'length_in_months'))
```

在视图中使用内联格式集

您可能需要提供一个视图，允许用户编辑模型的相关对象。以下是如何做到这一点：

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book, fields=('title',))
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something. Should generally end with a redirect. For example:
            return HttpResponseRedirect(author.get_absolute_url())
    else:
        formset = BookInlineFormSet(instance=author)
    return render_to_response("manage_books.html", {
        "formset": formset,
    })
```

注意我们如何在 `POST` 和 `GET` 例中传递 `instance` 。

指定要在内联表单中使用的窗口小部件

`inlineformset_factory` 使用 `modelformset_factory` 并将其大部分参数传递给 `modelformset_factory` 。这意味着您可以使用 `widgets` 参数，将其传递到 `modelformset_factory` 。请参阅上面的指定要在窗体中使用的窗口小部件的窗口小部件。

表单素材(`Media` 类)

渲染有吸引力的、易于使用的web表单不仅仅需要HTML -- 同时也需要CSS样式表，并且，如果你打算使用奇妙的web2.0组件，你也需要在每个页面包含一些JavaScript。任何提供的页面都需要CSS和JavaScript的精确配合，它依赖于页面上所使用的组件。

这就是素材定义所导入的位置。Django允许你将一些不同的文件 -- 像样式表和脚本 -- 与需要这些素材的表单和组件相关联。例如，如果你想要使用日历来渲染DateField，你可以定义一个自定义的日历组件。这个组件可以与渲染日历所需的CSS和JavaScript关联。当日历组件用在表单上的时候，Django可以识别出所需的CSS和JavaScript文件，并且提供一个文件名的列表，以便在你的web页面上简单地包含这些文件。

素材和Django Admin

Django的Admin应用为日历、过滤选择等一些东西定义了一些自定义的组件。这些组件定义了素材的需求，DJango Admin使用这些自定义组件来代替Django默认的组件。Admin模板只包含在提供页面上渲染组件所需的那些文件。

如果你喜欢Django Admin应用所使用的那些组件，可以在你的应用中随意使用它们。它们位于 `django.contrib.admin.widgets` 。

选择哪个JavaScript工具包？

现在有许多JavaScript工具包，它们中许多都包含组件（比如日历组件），可以用于提升你的应用。Django 有意避免去称赞任何一个JavaScript工具包。每个工具包都有自己的有点和缺点 -- 要使用适合你需求的任何一个。Django 有能力集成任何JavaScript工具包。

作为静态定义的素材

定义素材的最简单方式是作为静态定义。如果使用这种方式，定义在 `Media` 内部类中出现，内部类的属性定义了需求。

这是一个简单的例子：

```
from django import forms

class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

上面的代码定义了 `CalendarWidget` ，它继承于 `TextInput` 。每次CalendarWidget在表单上使用时，表单都会包含CSS文件 `pretty.css` ，以及JavaScript文件 `animations.js` 和 `actions.js` 。

静态定义在运行时被转换为名为 `media` 的组件属性。 `CalendarWidget` 实例的素材列表可以通过这种方式获取：

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/an
imations.js"></script>
<script type="text/javascript" src="http://static.example.com/ac
tions.js"></script>
```

下面是所有可能的 `Media` 选项的列表。它们之中没有必需选项。

css

各种表单和输出媒体所需的，描述CSS的字典。

字典中的值应该为文件名称的列表或者元组。对于如何指定这些文件的路径，详见路径的章节。

字典中的键位输出媒体的类型。它们和媒体声明中CSS文件接受的类型相同：'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' 和'tv'。如果你需要为不同的媒体类型使用不同的样式表，要为每个输出媒体提供一个CSS文件的列表。下面的例子提供了两个CSS选项 -- 一个用于屏幕，另一个用于打印：

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

如果一组CSS文件适用于多种输出媒体的类型，字典的键可以为输出媒体类型的逗号分隔的列表。在下面的例子中，TV和投影仪具有相同的媒体需求：

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

如果最后的CSS定义即将被渲染，会变成下面的HTML：

```
<link href="http://static.example.com/pretty.css" type="text/css
" media="screen" rel="stylesheet" />
<link href="http://static.example.com/lo_res.css" type="text/css
" media="tv,projector" rel="stylesheet" />
<link href="http://static.example.com/newspaper.css" type="text/
css" media="print" rel="stylesheet" />
```

js

所需的JavaScript文件由一个元组来描述。如何制定这些文件的路径，详见路径一节。

extend

一直布尔值，定义了 `Media` 声明的继承行为。

通常，任何使用静态 `Media` 定义的对象都会继承所有和父组件相关的素材。无论父对象如何定义它自己的需求，都是这样。例如，如果我们打算从上面的例子中扩展我们的基础日历控件：

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
<link href="http://static.example.com/fancy.css" type="text/css"
 media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/an
imations.js"></script>
<script type="text/javascript" src="http://static.example.com/ac
tions.js"></script>
<script type="text/javascript" src="http://static.example.com/wh
izbang.js"></script>
```

FancyCalendar 组件继承了所有父组件的素材。如果你不想让 `Media` 以这种方式被继承，要向 `Media` 声明中添加 `extend=False` 声明：

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/fancy.css" type="text/css"
 media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/wh
izbang.js"></script>
```

如果你需要对继承进行更多控制，要使用动态属性来定义你的素材。动态属性可以提供更多的控制，来控制继承哪个文件。

Media as a dynamic property

如果你需要对素材需求进行更多的复杂操作，你可以直接定义 `media` 属性。这可以通过定义一个返回 `forms.Media` 实例的组件属性来实现。 `forms.Media` 的构造器接受 `css` 和 `js` 关键字参数，和在静态媒体定义中的格式相同。

例如，我们的日历组件的静态定义可以定义成动态形式：

```
class CalendarWidget(forms.TextInput):
    def _media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
    media = property(_media)
```

对于如何构建动态 `media` 属性的的返回值，详见媒体对象一节。

素材定义中的路径

用于指定素材的路径可以是相对的或者绝对的。如果路径以 `/` ， `http://` 或者 `https://` 开头，会被解释为绝对路径。所有其它的路径会在开头追加合适前缀的值。

作为 *staticfiles app*的简介的一部分，添加了两个新的设置，它们涉及到渲染完整页面所需的“静态文件”： `STATIC_URL` 和 `STATIC_ROOT` 。

Django 会检查是否 `STATIC_URL` 设置不是 `None`，来寻找合适的前缀来使用，并且会自动回退使用 `MEDIA_URL`。例如，如果你站点的 `MEDIA_URL` 是 `'http://uploads.example.com/'` 并且 `STATIC_URL` 是 `None`：

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('/css/pretty.css',),
...         }
...         js = ('animations.js', 'http://othersite.com/actions.js')

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://uploads.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

但如果 `STATIC_URL` 为 `'http://static.example.com/'`：

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

`Media` 对象

当你访问表单上的一个组件的 `media` 属性时，返回值是一个 `forms.Media` 对象。就像已经看到的那样，表示 `Media` 对象的字符串，是在你的HTML页面的 `<head>` 代码段包含相关文件所需的HTML。

然而，`Media` 对象具有一些其它的有趣属性。

素材的子集

如果你仅仅想得到特定类型的文件，你可以使用下标运算符来过滤出你感兴趣的媒体。例如：

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/an
imations.js"></script>
<script type="text/javascript" src="http://static.example.com/ac
tions.js"></script>

>>> print(w.media['css'])
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
```

当你使用下标运算符的时候，返回值是一个新的 `Media` 对象，但是只含有感兴趣的媒体。

组合 `Media` 对象

`Media` 对象可以添加到一起。添加两个 `Media` 的时候，产生的 `Media` 对象含有二者指定的素材的并集：

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('pretty.css',)
...         }
...         js = ('animations.js', 'actions.js')

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ('whizbang.js',)

>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/an
imations.js"></script>
<script type="text/javascript" src="http://static.example.com/ac
tions.js"></script>
<script type="text/javascript" src="http://static.example.com/wh
izbang.js"></script>
```

表单上的 `Media`

组件并不是唯一拥有`media`定义的对象 -- 表单可以定义`media`。在表单上定义`media`的规则和组件上面一样：定义可以为静态的或者动态的。声明的路径和继承规则也严格一致。

无论是否你定义了`media`，所有表单对象都有`media`属性。这个属性的默认值是，向所有属于这个表单的组件添加`media`定义的结果。

```
>>> from django import forms
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/an
imations.js"></script>
<script type="text/javascript" src="http://static.example.com/ac
tions.js"></script>
<script type="text/javascript" src="http://static.example.com/wh
izbang.js"></script>
```

如果你打算向表单关联一些额外的素材 -- 例如，表单布局的CSS -- 只是向表单添加`Media`声明就可以了：

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...
...     class Media:
...         css = {
...             'all': ('layout.css',)
...         }

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css
" media="all" rel="stylesheet" />
<link href="http://static.example.com/layout.css" type="text/css
" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/an
imations.js"></script>
<script type="text/javascript" src="http://static.example.com/ac
tions.js"></script>
<script type="text/javascript" src="http://static.example.com/wh
izbang.js"></script>
```

> 译者：Django 文档协作翻译小组，原文：Integrating media。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

表单集

class `BaseFormSet` [source]

表单集是同一个页面上多个表单的抽象。它非常类似于一个数据表格。假设有下述表单：

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

你可能希望允许用户一次创建多个Article。你可以根据 `ArticleForm` 创建一个表单集：

```
>>> from django.forms.formsets import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

你已经创建一个命名为 `ArticleFormSet` 的表单集。表单集让你能迭代表单集中的表单并显示它们，就和普通的表单一样：

```
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><inp
ut type="text" name="form-0-title" id="id_form-0-title" /></td><
/tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><t
d><input type="text" name="form-0-pub_date" id="id_form-0-pub_da
te" /></td></tr>
```

正如你所看到的，这里仅显示一个空表单。显示的表单的数目通过 `extra` 参数控制。默认情况下，`formset_factory()` 定义一个表单；下面的示例将显示两个空表单：

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

对 `formset` 的迭代将以它们创建时的顺序渲染表单。通过提供一个 `__iter__()` 方法，可以改变这个顺序。

表单集还可以索引，它将返回对应的表单。如果覆盖 `__iter__` ，你还需要覆盖 `__getitem__` 以获得一致的行为。

表单集的初始数据

初始数据体现着表单集的主要功能。如上所述，你可以定义表单的数目。它表示除了从初始数据生成的表单之外，还要生成多少个额外的表单。让我们看个例子：

```
>>> import datetime
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Django is now open source',
...      'pub_date': datetime.date.today(),}
... ])

>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><inp
ut type="text" name="form-0-title" value="Django is now open sou
rce" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><t
d><input type="text" name="form-0-pub_date" value="2008-05-12" i
d="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><inp
ut type="text" name="form-1-title" id="id_form-1-title" /></td><
/tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><t
d><input type="text" name="form-1-pub_date" id="id_form-1-pub_da
te" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><inp
ut type="text" name="form-2-title" id="id_form-2-title" /></td><
/tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><t
d><input type="text" name="form-2-pub_date" id="id_form-2-pub_da
te" /></td></tr>
```

上面现在一共有三个表单。一个是初始数据生成的，还有两个是额外的表单。还要注意的是，我们传递的初始数据是一个由字典组成的列表。

另见

利用模型表单集从模型中创建表单集。

限制表单的最大数量

`formset_factory()` 的 `max_num` 参数 ，给予你限制表单集展示表单个数的能力

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_n
um=1)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><inp
ut type="text" name="form-0-title" id="id_form-0-title" /></td><
/tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><t
d><input type="text" name="form-0-pub_date" id="id_form-0-pub_da
te" /></td></tr>
```

假如 `max_num` 的值 比已经在初始化数据中存在的条目数目多的话, `extra` 对应个数的额外空表单将会被添加到表单集， 只要表单总数不超过 `max_num` . 例如，如果 `extra=2` 和 `max_num=2` ，并且用一个 `initial` 项初始化表单集，将显示空白表单。

假如初始化数据的条目超过 `max_num` 的值, 所有初始化数据表单都会被展现并且忽视 `max_num` 值的限定 ，而且不会有额外的表单被呈现。比如, 如果 `extra=3` ， `max_num=1` 并且表单集由两个初始化条窗，那么两个带有初始化数据的表单将被呈现。

`max_num` 的值为 `None` (默认值) 等同于限制了一个比较高的展现表单数目(1000个). 实际上就是等同于没限制.

默认的, `max_num` 只影响了表单的数目展示，但不影响验证. 假如 `validate_max=True` 传给了 `formset_factory()` , 然后 `max_num` 才将会影响验证. 请参阅*Validating the number of forms in a formset*。

表单验证

表单集的验证几乎和 一般的 `Form` 一样. 表单集里面有一个 `is_valid` 的方法来提供快捷的验证所有表单的功能。

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm)
>>> data = {
...     'form-TOTAL_FORMS': '1',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
True
```

我们没有传递任何数据到formset，导致一个有效的形式。表单集足够聪明，可以忽略未更改的其他表单。如果我们提供无效的文章：

```
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test',
...     'form-1-pub_date': '', # <-- this date is missing but re
quired
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
```

正如我们看见的，`formset.errors` 是一个列表， 他包含的错误信息正好与表单集内的表单一一对应错误检查会在两个表单中分别执行，被预见的错误出现错误列表的第二项

`BaseFormSet.``total_error_count` ()[source]

想知道表单集内有多少个错误可以使用 `total_error_count` 方法

```
>>> # Using the previous example
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
>>> len(formset.errors)
2
>>> formset.total_error_count()
1
```

我们也可以检查表单数据是否从初始值发生了变化 (i.e. the form was sent without any data):

```
>>> data = {
...     'form-TOTAL_FORMS': '1',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': '',
...     'form-0-pub_date': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.has_changed()
False
```

了解ManagementForm

你也许已经注意到了那些附加的数据 (`form-TOTAL_FORMS` , `form-INITIAL_FORMS` and `form-MAX_NUM_FORMS`) 他们是必要的，且必须位于表单集数据的最上方 这些必须传递给 `ManagementForm` . ManagementFormThis 用于管理表单集中的表单. 如果你不提供这些数据，将会触发异常

```
>>> data = {
...     'form-0-title': 'Test',
...     'form-0-pub_date': '',
... }
>>> formset = ArticleFormSet(data)
Traceback (most recent call last):
...
django.forms.utils.ValidationError: ['ManagementForm data is mis
sing or has been tampered with']
```

也同样用于记录多少的表单实例将被展示如果您通过JavaScript添加新表单，则应该增加此表单中的计数字段。On the other hand, if you are using JavaScript to allow deletion of existing objects, then you need to ensure the ones being removed are properly marked for deletion by including `form-#-DELETE` in the `POST` data. 期望所有形式存在于 `POST` 数据中。

管理表单可用作表单集本身的属性。在模板中呈现表单集时，您可以通过呈现 `{{ my_formset.management_form }}` t0>（替换您的formset的名称适当）。

total_form_count and

`BaseFormSet` 有一些与 `ManagementForm` ， `total_form_count` 和 `initial_form_count` 密切相关的方法。

`total_form_count` 返回此表单集中的表单总数。 `initial_form_count` 返回Formset中预填充的表单数，也用于确定需要多少表单。你可能永远不需要重写这些方法，所以请确保你明白他们做什么之前这样做。

empty_form

`BaseFormSet` 提供了一个附加属性 `empty_form` ，它返回一个前缀为 `__prefix__` 的表单实例，以便于使用JavaScript的动态表单。

自定义表单集验证

A formset has a `clean` method similar to the one on a `Form` class. 这是您定义自己的验证，在formset级别工作：

```
>>> from django.forms.formsets import BaseFormSet
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):
...     def clean(self):
...         """Checks that no two articles have the same title."
""
...         if any(self.errors):
...             # Don't bother validating the formset unless eac
h form is valid on its own
...             return
...         titles = []
...         for form in self.forms:
...             title = form.cleaned_data['title']
...             if title in titles:
...                 raise forms.ValidationError("Articles in a s
et must have distinct titles.")
...             titles.append(title)

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseAr
ticleFormSet)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Articles in a set must have distinct titles.']
```

在所有 `Form.clean` 方法被调用后，调用formset `clean` 方法。将使用表单集上的 `non_form_errors()` 方法找到错误。

验证表单集中的表单数

Django 提供了两种方法去检查表单能够提交的最大数和最小数，应用如果需要更多的关于提交数量的自定义验证逻辑，应该使用自定义表单击验证

validate_max

I如果 `validate_max=True` 被提交给 `formset_factory()` , validation 将在数据集中检查被提交表单的数量, 减去被标记删除的, 必须小于等于 `max_num` .

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, max_num=1, val
idate_max=True)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MIN_NUM_FORMS': '',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test 2',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Please submit 1 or fewer forms.']
```

`validate_max=True` validates 将会对 `max_num` 严格限制，即使提供的初始数据超过 `max_num` 而导致其无效

注意

Regardless of `validate_max` , if the number of forms in a data set exceeds `max_num` by more than 1000, then the form will fail to validate as if `validate_max` were set, and additionally only the first 1000 forms above `max_num` will be validated. 剩余部分将被完全截断。这是为了防止使用伪造的POST请求的内存耗尽攻击。

validate_min

New in Django 1.7.

如果 `validate_min=True` 被传递到 `formset_factory()` ，验证也将检查数据集中的表格数量减去那些被标记为删除的表格数量大于或等于到 `min_num` 。

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, min_num=3, val
idate_min=True)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MIN_NUM_FORMS': '',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test 2',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Please submit 3 or more forms.']
```

Changed in Django 1.7:

将 `min_num` 和 `validate_min` 参数添加到 `formset_factory()` 中。

表单的排序和删除行为

`formset_factory()` 提供两个可选参数 `can_order` 和 `can_delete` 来实现表单集中表单的排序和删除。

can_order

`BaseFormSet.``can_order`

默认值： `False`

使你创建能排序的表单集。

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_order=True
)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008,
5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008,
5, 11)},
... ])
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><inp
ut type="text" name="form-0-title" value="Article #1" id="id_for
m-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><t
d><input type="text" name="form-0-pub_date" value="2008-05-10" i
d="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-ORDER">Order:</label></th><td><inp
ut type="number" name="form-0-ORDER" value="1" id="id_form-0-ORD
ER" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><inp
ut type="text" name="form-1-title" value="Article #2" id="id_for
m-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><t
d><input type="text" name="form-1-pub_date" value="2008-05-11" i
d="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-1-ORDER">Order:</label></th><td><inp
ut type="number" name="form-1-ORDER" value="2" id="id_form-1-ORD
ER" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><inp
ut type="text" name="form-2-title" id="id_form-2-title" /></td><
/tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><t
d><input type="text" name="form-2-pub_date" id="id_form-2-pub_da
te" /></td></tr>
<tr><th><label for="id_form-2-ORDER">Order:</label></th><td><inp
ut type="number" name="form-2-ORDER" id="id_form-2-ORDER" /></td
></tr>
```

它会给每个表单添加一个字段，新字段命名 `ORDER` 并且是 `forms.IntegerField` 类型。它根据初始数据，为这些表单自动生成数值。下面让我们看一下，如果用户改变这个值会发生什么变化:

```
>>> data = {
...     'form-TOTAL_FORMS': '3',
...     'form-INITIAL_FORMS': '2',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Article #1',
...     'form-0-pub_date': '2008-05-10',
...     'form-0-ORDER': '2',
...     'form-1-title': 'Article #2',
...     'form-1-pub_date': '2008-05-11',
...     'form-1-ORDER': '1',
...     'form-2-title': 'Article #3',
...     'form-2-pub_date': '2008-05-01',
...     'form-2-ORDER': '0',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008,
5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008,
5, 11)},
... ])
>>> formset.is_valid()
True
>>> for form in formset.ordered_forms:
...     print(form.cleaned_data)
{'pub_date': datetime.date(2008, 5, 1), 'ORDER': 0, 'title': 'Ar
ticle #3'}
{'pub_date': datetime.date(2008, 5, 11), 'ORDER': 1, 'title': 'A
rticle #2'}
{'pub_date': datetime.date(2008, 5, 10), 'ORDER': 2, 'title': 'A
rticle #1'}
```

can_delete

`BaseFormSet.``can_delete`

默认值： `False`

使你创建一个表单集，可以选择删除一些表单。

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_delete=Tru
e)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008,
5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008,
5, 11)},
... ])
>>> for form in formset:
....    print(form.as_table())
<input type="hidden" name="form-TOTAL_FORMS" value="3" id="id_fo
rm-TOTAL_FORMS" /><input type="hidden" name="form-INITIAL_FORMS"
 value="2" id="id_form-INITIAL_FORMS" /><input type="hidden" nam
e="form-MAX_NUM_FORMS" id="id_form-MAX_NUM_FORMS" />
<tr><th><label for="id_form-0-title">Title:</label></th><td><inp
ut type="text" name="form-0-title" value="Article #1" id="id_for
m-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><t
d><input type="text" name="form-0-pub_date" value="2008-05-10" i
d="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-DELETE">Delete:</label></th><td><i
nput type="checkbox" name="form-0-DELETE" id="id_form-0-DELETE"
/></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><inp
ut type="text" name="form-1-title" value="Article #2" id="id_for
m-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><t
d><input type="text" name="form-1-pub_date" value="2008-05-11" i
d="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-1-DELETE">Delete:</label></th><td><i
nput type="checkbox" name="form-1-DELETE" id="id_form-1-DELETE"
/></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><inp
ut type="text" name="form-2-title" id="id_form-2-title" /></td><
/tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><t
d><input type="text" name="form-2-pub_date" id="id_form-2-pub_da
te" /></td></tr>
<tr><th><label for="id_form-2-DELETE">Delete:</label></th><td><i
nput type="checkbox" name="form-2-DELETE" id="id_form-2-DELETE"
/></td></tr>
```

与 `can_order` 类似，它添加一个名为 `DELETE` 的新字段，并且是 `forms.BooleanField` 类型。如下，你可以通过 `deleted_forms` 来获取标记删除字段的数据：

```
>>> data = {
...     'form-TOTAL_FORMS': '3',
...     'form-INITIAL_FORMS': '2',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Article #1',
...     'form-0-pub_date': '2008-05-10',
...     'form-0-DELETE': 'on',
...     'form-1-title': 'Article #2',
...     'form-1-pub_date': '2008-05-11',
...     'form-1-DELETE': '',
...     'form-2-title': '',
...     'form-2-pub_date': '',
...     'form-2-DELETE': '',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008,
5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008,
5, 11)},
... ])
>>> [form.cleaned_data for form in formset.deleted_forms]
[{'DELETE': True, 'pub_date': datetime.date(2008, 5, 10), 'title
': 'Article #1'}]
```

如果你使用 `ModelFormSet`，调用 `formset.save()` 将删除那些有删除标记的表单的模型实例。

Changed in Django 1.7:

如果你调用 `formset.save(commit=False)`，对像将不会被自动删除。你需要调用 `formset.deleted_objects` 每个对像的 `delete()` 来真正删除他们。

```
>>> instances = formset.save(commit=False)
>>> for obj in formset.deleted_objects:
...     obj.delete()
```

如果你想保持向前兼容 Django 1.6 或更早的版本，你需要这样做：

```
>>> try:
>>>     # For Django 1.7+
>>>     for obj in formset.deleted_objects:
>>>         obj.delete()
>>> except AssertionError:
>>>     # Django 1.6 and earlier already deletes the objects, tr
ying to
>>>     # delete them a second time raises an AssertionError.
>>>     pass
```

On the other hand, if you are using a plain `FormSet`, it's up to you to handle `formset.deleted_forms`, perhaps in your formset's `save()` method, as there's no general notion of what it means to delete a form.

给表单集添加一个额外的字段

如果你想往表单集中添加额外的字段，是十分容易完成的，表单集的基类（BaseFormSet）提供了 `add_fields` 方法。可以简单的通过重写这个方法来添加你自己的字段，甚至重新定义order和deletion字段的方法和属性：

```
>>> from django.forms.formsets import BaseFormSet
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     def add_fields(self, form, index):
...         super(BaseArticleFormSet, self).add_fields(form, ind
ex)
...         form.fields["my_field"] = forms.CharField()

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseAr
ticleFormSet)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><inp
ut type="text" name="form-0-title" id="id_form-0-title" /></td><
/tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><t
d><input type="text" name="form-0-pub_date" id="id_form-0-pub_da
te" /></td></tr>
<tr><th><label for="id_form-0-my_field">My field:</label></th><t
d><input type="text" name="form-0-my_field" id="id_form-0-my_fie
ld" /></td></tr>
```

在视图和模板中使用表单集

在视图中使用表单集就像使用标准的 `Form` 类一样简单，唯一要做的就是确信你在模板中处理表单。让我们看一个简单视图：

```
from django.forms.formsets import formset_factory
from django.shortcuts import render_to_response
from myapp.forms import ArticleForm

def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == 'POST':
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
            pass
    else:
        formset = ArticleFormSet()
    return render_to_response('manage_articles.html', {'formset'
: formset})
```

`manage_articles.html` 模板也可以像这样：

```
<form method="post" action="">
    {{ formset.management_form }}
    <table>
        {% for form in formset
        {{ form }}
        {% endfor </table>
</form>
```

不过，上面可以用一个快捷写法，让表单集来分发管理表单：

```
<form method="post" action="">
    <table>
        {{ formset }}
    </table>
</form>
```

上面表单集调用 `as_table` 方法。

手动渲染 `can_delete`

如果你在模板中手工渲染字段，那么渲染 `can_delete` 参数用 `{{ form.` DELETE }}：

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset
        <ul>
            <li>{{ form.title }}</li>
            <li>{{ form.pub_date }}</li>
            {% if formset.can_delete
                <li>{{ form.DELETE }}</li>
            {% endif
        </ul>
    {% endfor </form>
```

类似的，如果表单集有排序功能(`can_order=True`)，可以使用 `{{ form.ORDER }}`渲染。

在视图中使用多个表单集

可以在视图中使用多个表单集，表单集从表单中借鉴了很多方法你可以使用 `prefix` 给每个表单字段添加前缀，以允许多个字段传递给视图，而不发生命名冲突 让我们看看可以怎么做

```
from django.forms.formsets import formset_factory
from django.shortcuts import render_to_response
from myapp.forms import ArticleForm, BookForm

def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    BookFormSet = formset_factory(BookForm)
    if request.method == 'POST':
        article_formset = ArticleFormSet(request.POST, request.F
ILES, prefix='articles')
        book_formset = BookFormSet(request.POST, request.FILES,
prefix='books')
        if article_formset.is_valid() and book_formset.is_valid(
):
            # do something with the cleaned_data on the formsets
.
            pass
    else:
        article_formset = ArticleFormSet(prefix='articles')
        book_formset = BookFormSet(prefix='books')
    return render_to_response('manage_articles.html', {
        'article_formset': article_formset,
        'book_formset': book_formset,
    })
```

你可以以正常的方式渲染模板。记住 `prefix` 在POST请求和非POST 请求中均需设置，以便他能渲染和执行正确

表单验证和字段验证

表单验证发生在数据验证之后。如果你需要定制化这个过程，有几个不同的地方可以修改，每个地方的目的不一样。表单处理过程中要运行三种类别的验证方法。它们通常在你调用表单的 `is_valid()` 方法时执行。还有其它方法可以触发验证过程（访问 `errors` 属性或直接调用 `full_clean()` ），但是通用情况下不需要。

一般情况下，如果处理的数据有问题，每个类别的验证方法都会引发 `ValidationError` ，并将相关信息传递给 `ValidationError` 。参见下文中引发 `ValidationError` 的最佳实践。如果没有引发 `ValidationError` ，这些方法应该返回验证后的（规整化的）数据的Python 对象。

大部分应该可以使用validators 完成，它们可以很容易地重用。Validators 是简单的函数（或可调用对象），它们接收一个参数并对非法的输入抛出 `ValidationError` 。Validators 在字段的 `to_python` 和 `validate` 方法调用之后运行。

表单的验证划分成几个步骤，它们可以定制或覆盖：

- 字段的 `to_python()` 方法是验证的第一步。它将值强制转换为正确的数据类型，如果不能转换则引发 `ValidationError` 。 这个方法从Widget 接收原始的值并返回转换后的值。例如，FloatField 将数据转换为Python 的 `float` 或引发 `ValidationError` 。
- 字段的 `validate()` 方法处理字段特异性的验证，这种验证不适合位于validator 中。它接收一个已经转换成正确数据类型的值， 并在发现错误时引发 `ValidationError` 。这个方法不返回任何东西且不应该改变任何值。当你遇到不可以或不想放在validator 中的验证逻辑时，应该覆盖它来处理验证。
- 字段的 `run_validators()` 方法运行字段的所有Validator，并将所有的错误信息聚合成一个单一的 `ValidationError` 。你应该不需要覆盖这个方法。
- Field子类的 `clean()` 方法。它负责以正确的顺序运行 `to_python` 、 `validate` 和 `run_validators` 并传播它们的错误。如果任何时刻、任何方法引发 `ValidationError` ，验证将停止并引发这个错误。这个方法返回验证后的数据，这个数据在后面将插入到表单的 `cleaned_data` 字典中。
- 表单子类中的 `clean_<fieldname>()` 方法 —— `<fieldname>` 通过表单中的字段名称替换。这个方法完成于特定属性相关的验证，这个验证与字段的类型无关。这个方法没有任何传入的参数。你需要查找 `self.cleaned_data` 中该字段的值，记住此时它已经是一个Python 对象而不是表单中提交的原始字符串（它位于 `cleaned_data` 中是因为字段的 `clean()` 方法已经验证过一次数据）。

例如，如果你想验证名为 `serialnumber` 的 `CharField` 的内容是否唯一，`clean_serialnumber()` 将是实现这个功能的理想之处。你需要的不是一个特别的字段（它只是一个 `CharField` ），而是一个特定于表单字段特定验证，并规整化数据。

这个方法返回从cleaned_data 中获取的值，无论它是否修改过。

- 表单子类的 `clean()` 方法。这个方法可以实现需要同时访问表单多个字段的验证。这里你可以验证如果提供字段 `A` ，那么字段 `B` 必须包含一个合法的邮件地址以及类似的功能。 这个方法可以返回一个完全不同的字典，该字典将用作 `cleaned_data` 。

  因为字段的验证方法在调用 `clean()` 时会运行，你还可以访问表单的 `errors` 属性，它包含验证每个字段时的所有错误。

  注意，你覆盖的 `Form.clean()` 引发的任何错误将不会与任何特定的字段关联。它们位于一个特定的“字段”（叫做 `__all__` ）中，如果需要可以通过 `non_field_errors()` 方法访问。如果你想添加一个特定字段的错误到表单中，需要调用 `add_error()` 。

  还要注意，覆盖 `ModelForm` 子类的 `clean()` 方法需要特殊的考虑。（更多信息参见*ModelForm* 文档）。

这些方法按以上给出的顺序执行，一次验证一个字段。也就是说，对于表单中的每个字段（按它们在表单定义中出现的顺序），先运行 `Field.clean()` ，然后运行 `clean_<fieldname>()` 。每个字段的这两个方法都执行完之后，最后运行 `Form.clean()` 方法，无论前面的方法是否抛出过异常。

下面有上面每个方法的示例。

我们已经提到过，所有这些方法都可以抛出 `ValidationError` 。对于每个字段，如果 `Field.clean()` 方法抛出 `ValidationError` ，那么将不会调用该字段对应的clean_()方法。 但是，剩余的字段的验证方法仍然会执行。

抛出 `ValidationError`

为了让错误信息更加灵活或容易重写，请考虑下面的准则：

- 给构造函数提供一个富有描述性的错误码 `code` ：

```
# Good
    ValidationError(_('Invalid value'), code='invalid')

    # Bad
    ValidationError(_('Invalid value'))
```

- 不要预先将变量转换成消息字符串；使用占位符和构造函数的 `params` 参数：

```
# Good
    ValidationError(
        _('Invalid value: %(value)s'),
        params={'value': '42'},
    )

    # Bad
    ValidationError(_('Invalid value: %s') % value)
```

- 使用字典参数而不要用位置参数。这使得重写错误信息时不用考虑变量的顺序或者完全省略它们：

```
# Good
    ValidationError(
        _('Invalid value: %(value)s'),
        params={'value': '42'},
    )

    # Bad
    ValidationError(
        _('Invalid value: %s'),
        params=('42',),
    )
```

- 用 `gettext` 封装错误消息使得它可以翻译：

```
# Good
    ValidationError(_('Invalid value'))

    # Bad
    ValidationError('Invalid value')
```

所有的准则放在一起就是：

```
raise ValidationError(
    _('Invalid value: %(value)s'),
    code='invalid',
    params={'value': '42'},
)
```

如果你想编写可重用的表单、表单字段和模型字段，遵守这些准则是非常必要的。

如果你在验证的最后（例如，表单的 `clean()` 方法）且知道永远 不需要重新错误信息，虽然不提倡但你仍然可以选择重写不详细的信息：

```
ValidationError(_('Invalid value: %s') % value)
```

New in Django 1.7.

`Form.errors.as_data()` 和 `Form.errors.as_json()` 方法很大程度上受益于 `ValidationError` （利用 `code` 名和 `params` 字典）。

抛出多个错误

如果在一个验证方法中检查到多个错误并且希望将它们都反馈给表单的提交者，可以传递一个错误的列表给 `ValidationError` 构造函数。

和上面一样，建议传递的列表中的 `ValidationError` 实例都带有 `code` 和 `params` ，但是传递一个字符串列表也可以工作：

```
# Good
raise ValidationError([
    ValidationError(_('Error 1'), code='error1'),
    ValidationError(_('Error 2'), code='error2'),
])

# Bad
raise ValidationError([
    _('Error 1'),
    _('Error 2'),
])
```

实践验证

前面几节解释在一般情况下表单的验证是如何工作的。因为有时直接看功能在实际中的应用会更容易掌握，下面是一些列小例子，它们用到前面的每个功能。

使用Validator

Django 的表单（以及模型）字段支持使用简单的函数和类用于验证，它们叫做Validator。Validator 是可调用对象或函数，它接收一个值，如果该值合法则什么也不返回，否则抛出 `ValidationError` 。它们可以通过字段的 `validators` 参数传递给字段的构造函数，或者定义在 `Field` 类的 `default_validators` 属性中。

简单的Validator 可以用于在字段内部验证值，让我们看下Django 的 `SlugField` ：

```
from django.forms import CharField
from django.core import validators

class SlugField(CharField):
    default_validators = [validators.validate_slug]
```

正如你所看到的， `SlugField` 只是一个带有自定义Validator 的 `CharField` ，它们验证提交的文本符合某些字符规则。这也可以在字段定义时实现，所以：

```
slug = forms.SlugField()
```

等同于：

```
slug = forms.CharField(validators=[validators.validate_slug])
```

常见的情形，例如验证邮件地址和正则表达式，可以使用Django 中已经存在的Validator 类处理。例如， `validators.validate_slug` 是 `RegexValidator` 的一个实例，它构造时的第一个参数为： `^[-a-zA-Z0-9_]+$` 。编写*Validator* 一节可以查到已经存在的Validator 以及如何编写Validator 的一个示例。

表单字段的默认验证

让我们首先创建一个自定义的表单字段，它验证其输入是一个由逗号分隔的邮件地址组成的字符串。完整的类像这样：

```
from django import forms
from django.core.validators import validate_email

class MultiEmailField(forms.Field):
    def to_python(self, value):
        "Normalize data to a list of strings."

        # Return an empty list if no input was given.
        if not value:
            return []
        return value.split(',')

    def validate(self, value):
        "Check if value consists only of valid emails."

        # Use the parent's handling of required fields, etc.
        super(MultiEmailField, self).validate(value)

        for email in value:
            validate_email(email)
```

使用这个字段的每个表单都将在处理该字段数据之前运行这些方法。这个验证特定于该类型的字段，与后面如何使用它无关。

让我们来创建一个简单的 `ContactForm` 来向你演示如何使用这个字段：

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)
```

只需要简单地使用 `MultiEmailField` ，就和其它表单字段一样。当调用表单的 `is_valid()` 方法时， `MultiEmailField.clean()` 方法将作为验证过程的一部分运行，它将调用自定义的 `to_python()` 和 `validate()` 方法。

验证特定字段属性

继续上面的例子，假设在我们的 `ContactForm` 中，我们想确保 `recipients` 字段始终包含 `"fred@example.com"` 。这是特定于我们这个表单的验证，所以我们不打算将它放在通用的 `MultiEmailField` 类中。我们将编写一个运行在 `recipients` 字段上的验证方法，像这样：

```
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean_recipients(self):
        data = self.cleaned_data['recipients']
        if "fred@example.com" not in data:
            raise forms.ValidationError("You have forgotten abou
t Fred!")

        # Always return the cleaned data, whether you have chang
ed it or
        # not.
        return data
```

验证相互依赖的字段

假设我们添加另外一个需求到我们的联系人表单中：如果 `cc_myself` 字段为 `True` ，那么 `subject` 必须包含单词 `"help"` 。我们的这个验证包含多个字段，所以表单的 `clean()` 方法是个不错的地方。注意，我们这里讨论的是表单

的 `clean()` 方法，之前我们编写的字段的 `clean()` 方法。区别字段和表单之间的差别非常重要。字段是单个数据，表单是字段的集合。

在调用表单 `clean()` 方法的时候，所有字段的验证方法已经执行完（前两节），所以 `self.cleaned_data` 填充的是目前为止已经合法的数据。所以你需要记住这个事实，你需要验证的字段可能没有通过初试的字段检查。

在这一步，有两种方法报告错误。最简单的方法是在表单的顶端显示错误。你可以在 `clean()` 方法中抛出 `ValidationError` 来创建错误。例如：

```
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super(ContactForm, self).clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject:
            # Only do something if both fields are valid so far.
            if "help" not in subject:
                raise forms.ValidationError("Did not send for 'h
elp' in "
                        "the subject despite CC'ing yourself.")
```

Changed in Django 1.7:

在以前版本的Django 中，要求 `form.clean()` 返回 `cleaned_data` 的一个字典。现在，这个方法仍然可以返回将要用到的数据的字典，但是不再是强制的。

在这段代码中，如果抛出验证错误，表单将在表单的顶部显示（通常是）描述该问题的一个错误信息。

注意，示例代码中 `super(ContactForm, self).clean()` 的调用时为了保证维持父类中的验证逻辑。

第二种方法涉及将错误消息关联到某个字段。在这种情况下，让我们在表单的显示中分别关联一个错误信息到“subject” 和“cc_myself” 行。在实际应用中要小心，因为它可能导致表单的输出变得令人困惑。我们只是向你展示这里可以怎么做，在特定的情况下，需要你和你的设计人员确定什么是好的方法。我们的新代码（代替前面的示例）像这样：

```
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super(ContactForm, self).clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject and "help" not in subject:
            msg = "Must put 'help' in subject when cc'ing yourse
lf."
            self.add_error('cc_myself', msg)
            self.add_error('subject', msg)
```

`add_error()` 的第二个参数可以是一个简单的字符串，但更倾向是 `ValidationError` 的一个实例。更多细节参见抛出*ValidationError*。注意， `add_error()` 将从 `cleaned_data` 中删除相应的字段。

开发过程

学习各种组件和工具帮助你对Django应用进行改进和测试：

设置

Django 的设置

Django 的设置文件包含你安装的Django 的所有配置。这页文档解释设置是如何工作以及有哪些设置。

基础

设置文件只是一个Python 模块，带有模块级别的变量。

下面是一些示例设置：

```
ALLOWED_HOSTS = ['www.example.com']
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
```

> 注
>
> 如果你设置 `DEBUG` 为 `False`，那么你应该正确设置 `ALLOWED_HOSTS` 的值。

因为设置文件是一个Python 模块，所以适用以下情况：

- 不允许出现Python 语法错误。
- 它可以使用普通的Python 语法动态地设置。例如：

```
MY_SETTING = [str(i) for i in range(30)]
```

- 它可以从其它设置文件导入值。

指定设置文件

`DJANGO_SETTINGS_MODULE`

当你使用Django 时，你必须告诉它你正在使用哪个设置。这可以使用环境变量 `DJANGO_SETTINGS_MODULE` 来实现。

`DJANGO_SETTINGS_MODULE` 的值应该使用Python 路径的语法，例如 `mysite.settings` 。注意，设置模块应该在Python 的导入查找路径 中。

django-admin 工具

当使用 `django-admin` 时， 你可以设置只设置环境变量一次，或者每次运行该工具时显式传递设置模块。

例如（Unix Bash shell）：

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

例如（Windows shell）：

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

使用--settings 命令行参数可以手工指定设置：

```
django-admin runserver --settings=mysite.settings
```

在服务器上(mod_wsgi)

在线上服务器环境中，你需要告诉WSGI 的application 使用哪个设置文件。可以使用os.environ 实现：

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

阅读Django mod_wsgi 文档 以获得关于Django WSGI application 的更多和其它常见信息。

默认的设置

Django 的设置文件不需要定义所有的设置。每个设置都有一个合理的默认值。这些默认值位于 `django/conf/global_settings.py` 模块中。

下面是Django 用来编译设置的算法：

- 从 `global_settings.py` 中加载设置。
- 从指定的设置文件中加载设置，如有必要则覆盖全局的设置。

注意，设置文件不 应该从global_settings 中导入，因为这是多余的。

查看改变的设置

有一个简单的方法可以查看哪些设置与默认的设置不一样了。python `manage.py diffsettings` 命令显示当前的设置文件和Django 默认设置之间的差异。

获取更多信息，查看 `diffsettings` 的文档。

在Python 代码中使用设置

在Django 应用中，可以通过导入 `django.conf.settings` 对象来使用设置。例如：

```
from django.conf import settings

if settings.DEBUG:
    # Do something
```

注意， `django.conf.settings` 不是一个模块 —— 它是一个对象。所以不可以导入每个单独的设置：

```
from django.conf.settings import DEBUG  # This won't work.
```

还要注意，你的代码不应该 从 `global_settings` 或你自己的设置文件中导入。 `django.conf.settings` 抽象出默认设置和站点特定设置的概念；它表示一个单一的接口。它还可以将代码从你的设置所在的位置解耦出来。

运行时改变设置

请不要在应用运行时改变设置。例如，不要在视图中这样做：

```
from django.conf import settings

settings.DEBUG = True   # Don't do this!
```

给设置赋值的唯一地方是在设置文件中。

安全

因为设置文件包含敏感的信息，例如数据库密码，你应该尽一切可能来限制对它的访问。例如，修改它的文件权限使得只有你和Web 服务器使用者可以读取它。这在共享主机的环境中特别重要。

可用的设置

完整的可用设置清单，请参见设置参考。

创建你自己的设置

没有什么可以阻止你为自己的Django 应用创建自己的设置。只需要遵循下面的一些惯例：

- 设置名称全部是大写
- 不要使用一个已经存在的设置

对于序列类型的设置，Django 自己使用元组而不是列表，但这只是一个习惯。

不用DJANGO_SETTINGS_MODULE 设置

有些情况下，你可能想绕开 `DJANGO_SETTINGS_MODULE` 环境变量。例如，如果你正在使用自己的模板系统，而你不想建立指向设置模块的环境变量。

这些情况下，你可以手工配置Django 的设置。实现这点可以通过调用：

`django.conf.settings.configure(default_settings, **settings)`

例如：

```
from django.conf import settings

settings.configure(DEBUG=True)
```

可以传递 `configure()` 给任意多的关键字参数，每个关键字参数表示一个设置及其值。每个参数的名称应该都是大写，与上面讲到的设置名称相同。如果某个设置没有传递给 `configure()` 而且在后面需要使用到它，Django 将使用其默认设置的值。

当你在一个更大的应用中使用到Django 框架的一部分，有必要以这种方式配置Django —— 而且实际上推荐这么做。

所以，当通过 `settings.configure()` 配置时，Django 不会对进程的环境变量做任何修改（参见 `TIME_ZONE` 文档以了解为什么会发生）。在这些情况下，它假设你已经完全控制你的环境变量。

自定义默认的设置

如果你想让默认值来自其它地方而不是 `django.conf.global_settings` ，你可以传递一个提供默认设置的模块或类作为 `default_settings` 参数（或第一个位置参数）给 `configure()` 调用。

在下面的示例中，默认的设置来自 `myapp_defaults` ，并且设置 `DEBUG` 为 `True` ，而不论它在 `myapp_defaults` 中的值是什么：

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

下面的示例和上面一样，只是使用 `myapp_defaults` 作为一个位置参数：

```
settings.configure(myapp_defaults, DEBUG=True)
```

正常情况下，你不需要用这种方式覆盖默认值。Django 的默认值以及足够好使，你可以安全地使用它们。注意，如果你传递一个新的默认模块，你将完全取代 Django 的默认值，所以你必须指定每个可能用到的设置的值。完整的设置清单，参见 `django.conf.settings.global_settings` 。

configure() 和DJANGO_SETTINGS_MODULE 两者必居其一

如果你没有设置 `DJANGO_SETTINGS_MODULE` 环境变量，你 必须 在使用到读取设置的任何代码之前调用 `configure()` 。

如果你没有设置 `DJANGO_SETTINGS_MODULE` 且没有调用 `configure()` ，在首次访问设置时Django 将引发一个 `ImportError` 异常。

如果你设置了 `DJANGO_SETTINGS_MODULE` ，并访问了一下设置，然后 调用 `configure()` ，Django 将引发一个 `RuntimeError` 表示该设置已经有配置。有个属性正好可以用于这个情况：

例如：

```
from django.conf import settings
if not settings.configured:
    settings.configure(myapp_defaults, DEBUG=True)
```

另外，多次调用 `configure()` 或者在设置已经访问过之后调用 `configure()` 都是错误的。

归结为一点：只使用 `configure()` 或 `DJANGO_SETTINGS_MODULE` 中的一个。不可以两个都用和都不用。

> 另见
>
> 设置参考 包含完整的核心设置和contrib 应用设置的列表。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

设置

- 核心配置
- 权限
- 消息
- 会话
- 站点
- 静态文件
- 核心设置和主题索引

警告

当改变设置的时候你一定要小心，尤其当默认值是一个非空元组或者一个字典的时候，比如 `MIDDLEWARE_CLASSES` 和 `STATICFILES_FINDERS` . 确保组件符合Django的特性，你想使用的话。

核心配置

这里是一些Django的核心设置和它们的默认值。由contrib apps提供的设置，它的主题索引在下面列出。对于介绍材料， 请看 *settings topic guide*.

ABSOLUTE_URL_OVERRIDES

Default: `{}` (默认为空的字典)

这个字典把 `"app_label.model_name"` 字符串映射到带着一个模型对象的函数上并返回一个URL。这是一种在每次安装的基础上插入或覆盖 `get_absolute_url()` 方法的方法。例：

```
ABSOLUTE_URL_OVERRIDES = {
    'blogs.weblog': lambda o: "/blogs/%s/" % o.slug,
    'news.story': lambda o: "/stories/%s/%s/" % (o.pub_year, o.s
lug),
}
```

注意这里设置中使用的模型对象名称一定要小写，与模型的类名的实际情况无关

Changed in Django 1.7.1:

`ABSOLUTE_URL_OVERRIDES` 现在适用于未声明 `get_absolute_url()` 的模型。

ADMINS

默认值： `()` （空元组）

一个错误码消息数组当 `DEBUG=False` ，并且一个视图引发了异常, Django 会给这些人发一封含有完整异常信息的电子邮件。元组的每个成员应该是一个(姓名全称，电子邮件地址)的元组 . 例：

```
(('John', 'john@example.com'), ('Mary', 'mary@example.com'))
```

注意无论何时发生错误Django都会向这里的所有人发送邮件*all* 。查看*Error reporting* 了解更多信息。

ALLOWED_HOSTS

默认值： `[]` （空列表）

一个可以被Django站点提供提供服务的主机/域名的字符串名单。这是一个防御攻击者的措施，攻击会来源于缓存中毒然后密码被重置，并通过提交一个伪造了 `Host` header(主机头信息)的密码重置请求使得邮箱被链接到恶意主机，这是有可能发生的，即使在很多看似安全的web服务器配置中。

列表中的值要是完全合格的名称 (e.g. `'www.example.com'`), 这种情况下，他们将会正确地匹配请求的 `Host` header (忽略大小写，不包括端口).。开始处的英文句号能够用于作为子域名的通配符: `'.example.com'` 会匹配 `example.com` , `www.example.com` , 以及任何 `example.com` . 的子域名。 `'*'` 会匹配任何的值; 在这种情况中，你务必要提供你自己的 `Host` header的验证 (也可以是在中间件中，如果这样的话，中间件要首先被列入在 `MIDDLEWARE_CLASSES` 中).。

Changed in Django 1.7:

在先前的Django的版本中，如果你打算也要遵循fully qualified domain name (FQDN)(完全合格的域名名称), 一些浏览器会发送在 `Host` header(头标信息)中, 你要显示地添加另一个英文句号结尾的 `ALLOWED_HOSTS` .条目。这个条目也可以是个子域名通配符：

```
ALLOWED_HOSTS = [
    '.example.com',  # Allow domain and subdomains
    '.example.com.',  # Also allow FQDN and subdomains
]
```

在Django1.7中，末尾的点在执行主机验证的时候是被去掉的，因此一个条目没必要在末尾带点。

如果 `Host` header(头信息)(或者是 `X-Forwarded-Host` 如果 `USE_X_FORWARDED_HOST` 被启用的话) 不匹配这个列表中的任何值, 那么 `django.http.HttpRequest.get_host()` 函数将会抛出一个 `SuspiciousOperation` 异常.

当 `DEBUG` 的值为 `True` 或者运行测试程序的时候，host validation(主机认证)会被停用；任何主机都会被接受。因此通常只在生产环境中有必要这么设置。

这个validation(验证)只应用于 `get_host()`；如果你的代码是直接从 `request.META`访问 `Host` 头信息，那么你就绕过了安全保护措施。

ALLOWED_INCLUDE_ROOTS

默认值： `()` （空元组）

自1.8版起已弃用：不赞成使用这个设置和 `ssi` 模板标签，而且还会在Django2.0中被删除。

Changed in Django 1.8:

你可以在 `DjangoTemplates` 后台的 `OPTIONS` 中设置这个 `'allowed_include_roots'` 选项来代替。

这tuple(元组)中的字符串允许在模板中使用的时候加个 `{% ssi %}` 模板标签前缀。这是一个安全措施，所以编写模板的人不能访问那些不应该被访问的文件。

例如, 如果 `ALLOWED_INCLUDE_ROOTS` 的值为 `('/home/html', '/var/www')` ，那么模板中这样写： `{% ssi /home/html/foo.txt %}` 是没问题的 ，而像这样： `{% ssi /etc/passwd %}` 就不行了。（因为后者访问了/home/html目录以外的文件，这就是上面说的安全措施。译者注)

APPEND_SLASH

默认值： `True`

当设定为 `True` 时，如果请求的URL 没有匹配URLconf 里面的任何URL 并且没有以/（斜杠）结束，将重定向到以/ 结尾的URL。需要注意的是任何重定向都有可能导致post数据的丢失。

`APPEND_SLASH` 设置只有在安装了 `CommonMiddleware` 时才用到（参见中间件）。另见 `PREPEND_WWW` 。

高速缓存

默认：

```
{
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCach
e',
    }
}
```

一个字典包含所有缓存要使用的设置它是一个嵌套字典，其内容将高速缓存别名映射到包含单个高速缓存的选项的字典中。

`CACHES` 设置必须配置 `‘default’` 缓存；还可以指定任何数量的附加高速缓存。如果您正在使用本地内存高速缓存之外的其他高速缓存后端，或者需要定义多个高速缓存，这就需要添加其他高速缓存项。以下高速缓存选项可用。

后退

默认值：`''`（空字符串）

要使用的缓存后端。内置高速缓存后端是：

- `'django.core.cache.backends.db.DatabaseCache'`
- `'django.core.cache.backends.dummy.DummyCache'`
- `'django.core.cache.backends.filebased.FileBasedCache'`
- `'django.core.cache.backends.locmem.LocMemCache'`
- `'django.core.cache.backends.memcached.MemcachedCache'`
- `'django.core.cache.backends.memcached.PyLibMCCache'`

通过将 `BACKEND` 设置为缓存后端类的完全限定路径（即 `mypackage.backends.whatever.WhateverCache`），您可以使用未随Django提供的缓存后端。）。

KEY_FUNCTION

包含函数（或任何可调用）的虚线路径的字符串，定义如何将前缀，版本和键组成最终缓存键。默认实现等效于以下函数：

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), key])
```

你可以使用任何你想要的键功能，只要它有相同的参数签名。

有关详细信息，请参阅*cache documentation*。

KEY_PREFIX

默认值：`''`（空字符串）

一个字符串，将被自动包括（默认情况下预置）到Django服务器使用的所有缓存键。

有关详细信息，请参阅*cache documentation*。

LOCATION

默认值：`''`（空字符串）

要使用的缓存的位置。这可能是文件系统缓存的目录，内存缓存服务器的主机和端口，或者只是本地内存缓存的标识名称。例如：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBas
edCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

OPTIONS

默认值：无

额外的参数传递到缓存后端。可用参数因缓存后端而异。

有关可用参数的一些信息，请参见*Cache Backends*文档。有关详细信息，请参阅后端模块自己的文档。

TIMEOUT

默认值：300

高速缓存的有效时间。

New in Django 1.7.

如果此设置的值为 `空` ，则缓存将不会过期。

VERSION

默认值： `1`

Django服务器生成的缓存键的默认版本号。

有关详细信息，请参阅缓存文档。

CACHE_MIDDLEWARE_ALIAS

默认值： `default`

用于*cache middleware*的高速缓存连接。

CACHE_MIDDLEWARE_KEY_PREFIX

默认值： `''` （空字符串）

将由*cache middleware*生成的高速缓存键前缀的字符串。此前缀与 `KEY_PREFIX` 设置组合；它不替代它。

请参阅*Django's cache framework*。

CACHE_MIDDLEWARE_SECONDS

默认值： `600`

缓存*cache middleware*的页面的默认秒数。

请参阅*Django's cache framework*。

CSRF_COOKIE_AGE

New in Django 1.7.

Default: `31449600` (默认时间为一年.单位为秒)

CSRF cookies有效期 ,单位为秒

设置长期过期时间的原因是为了避免在用户关闭浏览器或将页面加入书签，然后从浏览器缓存加载该页面的情况下出现问题。没有持久性cookie，在这种情况下表单提交将失败。

某些浏览器（特别是Internet Explorer）可能不允许使用持久性cookie，或者可能使cookie jar的索引在磁盘上损坏，从而导致CSRF保护检查失败（有时间歇性）。将此设置更改为 `None` 可使用基于会话的CSRF Cookie，这些cookie将Cookie保存在内存中，而不是在持久存储上。

CSRF_COOKIE_DOMAIN

默认值： `None`

设置CSRF cookie时要使用的域。This can be useful for easily allowing cross-subdomain requests to be excluded from the normal cross site request forgery protection.应将其设置为一个字符串，例如 `".example.com"` ，以允许来自一个子域的表单的POST请求被另一个子域提供的视图接受。

Please note that the presence of this setting does not imply that Django's CSRF protection is safe from cross-subdomain attacks by default - please see the *CSRF limitations* section.

CSRF_COOKIE_HTTPONLY

默认值： `False`

是否在CSRF Cookie上使用 `HttpOnly` 标志。如果设置为 `True` ，客户端JavaScript将无法访问CSRF Cookie。

这可以帮助防止恶意JavaScript绕过CSRF保护。如果启用此选项并需要使用Ajax请求发送CSRF令牌的值，那么JavaScript将需要从页面上的隐藏CSRF令牌表单输入中提取值，而不是从Cookie中提取值。

See `SESSION_COOKIE_HTTPONLY` for details on `HttpOnly`.

CSRF_COOKIE_NAME

默认值：`'csrftoken'`

要用于CSRF身份验证令牌的cookie的名称。这可以是任何你想要的。请参阅*Cross Site Request Forgery protection*。

CSRF_COOKIE_PATH

默认值：`'/'`

在CSRF cookie上设置的路径。这应该匹配您的Django安装的URL路径，或者是该路径的父级。

如果您有多个运行在相同主机名下的Django实例，这将非常有用。他们可以使用不同的cookie路径，每个实例只会看到自己的CSRF cookie。

CSRF_COOKIE_SECURE

默认值：`False`

是否为CSRF Cookie使用安全Cookie。如果此设置为`True`，则Cookie将被标记为“安全”，这意味着浏览器可以确保该Cookie仅在HTTPS连接下发送。

CSRF_FAILURE_VIEW

默认值：`'django.views.csrf.csrf_failure'`

当传入请求被CSRF保护拒绝时要使用的视图函数的虚线路径。该函数应该有这个签名：

```
def csrf_failure(request, reason="")
```

其中`reason`是指示请求被拒绝的原因的短消息（针对开发人员或记录，而不是最终用户）。请参阅*Cross Site Request Forgery protection*。

DATABASES

默认：`{}`（空字典）

一个字典，包含Django 将使用的所有数据库的设置。它是一个嵌套的字典，其内容为数据库别名到包含数据库选项的字典的映射。

DATABASES 设置必须配置一个 `default` 数据库；可以同时指定任何数目的额外数据库。

最简单的配置文件可能是使用SQLite 建立一个数据库。这可以使用以下配置：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase',
    }
}
```

当连接其他数据库后端，比如MySQL、Oracle 或PostgreSQL，必须提供更多的连接参数。关于如何指定其他的数据库类型，参见后面的 `ENGINE` 设置。下面的例子用于PostgreSQL：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

下面是更复杂的配置可能需要的选项：

ATOMIC_REQUESTS

默认： `False`

当需要将每个HTTP 请求封装在一个数据库事务中时，设置它为 `True` 。参见*将事务与HTTP 请求绑定*。

AUTOCOMMIT

默认： `True`

如果你需要*禁用Django 的事务管理*并自己实现，设置它为 `False` 。

ENGINE

默认： `''` （空字符串）

使用的数据库后端。内建的数据库后端有：

- `'django.db.backends.postgresql_psycopg2'`
- `'django.db.backends.mysql'`
- `'django.db.backends.sqlite3'`
- `'django.db.backends.oracle'`

你可以不使用Django 自带的数据库后端，通过设置 `ENGINE` 为一个合法的路径即可（例如 `mypackage.backends.whatever` ）。

HOST

默认： `''` （空字符串）

连接数据库时使用哪个主机。空字符串意味着采用localhost 作为主机。 SQLite 不需要这个选项。

如果其值以斜杠（ `'/'` ）开始并且你使用的是MySQL，MySQL 将通过Unix socket 连接。例如：

```
"HOST": '/var/run/mysql'
```

如果你使用的是MySQL 并且该值不是以斜杠开头，那么将假设该值为主机。

如果你使用的是PostgreSQL，默认情况下（空 `HOST` ），数据库的连接通过UNIX domain sockets（ `pg_hba.conf` 中的'local'行）。如果你的UNIX domain socket 不在标准的路径，则使用 `postgresql.conf` 中的 `unix_socket_directory` 值。如果你想通过TCP sockets 连接，请设置 `HOST` 为'localhost' 或 '127.0.0.1'（ `pg_hba.conf` 中的'host'行）。在Windows上，你应该始终定义 `HOST` ，因为其不可以使用UNIX domain sockets。

NAME

默认： `''` （空字符串）

使用的数据库名称。对于SQLite，它是数据库文件的完整路径。指定路径时，请始终使用前向的斜杠，即使在Windows 上（例如 `C:/homes/user/mysite/sqlite3.db` ）。

CONN_MAX_AGE

默认： `0`

数据库连接的存活时间，以秒为单位。 `0` 表示在每个请求结束时关闭数据库连接 —— 这是Django 的历史遗留行为， `None` 表示无限的持久连接。

OPTIONS

默认：`{}`（空字典）

连接数据库时使用的额外参数。可用的参数与你的数据库后端有关。

在 数据库后端的文档中可以找到可用的参数的一些信息。更多信息，参考后端模块自身的文档。

PASSWORD

默认：`''` （空字符串）

连接数据库时使用的密码。SQLite 不需要这个选项。

PORT (端口)

默认：`''` （空字符串）

连接数据库时使用的端口。空字符串表示默认的端口。SQLite 不需要这个选项。

USER

默认：`''` （空字符串）

连接数据库时使用的用户名。SQLite 不需要这个选项。

TEST

Changed in Django 1.7:

所有 `TEST` 子条目用作数据库设置字典中的独立条目，并具有 `TEST_` 前缀。为了向后兼容旧版本的Django，您可以定义两个版本的设置，只要它们匹配。Further, `TEST_CREATE` , `TEST_USER_CREATE` and `TEST_PASSWD` were changed to `CREATE_DB` , `CREATE_USER` and `PASSWORD` respectively.

Default: `{}`

用于测试数据库的一个设置字典；有关创建和使用测试数据库的更多详细信息，请参见测试数据库。以下条目可用：

CHARSET

Default: `None`

字符集设置用于指定数据库编码格式。该设置的值会直接传给数据库，所以它的格式是由指定的数据库来决定的。

该字段支持 PostgreSQL (`postgresql_psycopg2`) 和MySQL (`mysql`) 数据库。

COLLATION

Default: `None`

创建测试数据库时要使用的归类顺序。此值直接传递到后端，因此其格式是后端特定的。

仅支持 `mysql` 后端（有关详细信息，请参阅MySQL手册）。

DEPENDENCIES

默认值： `['default']` ，对于除了 `default` 之外的所有数据库，没有依赖关系。

数据库的创建顺序依赖性。有关详细信息，请参阅有关*controlling the creation order of test databases*的文档。

MIRROR

默认值： `None`

此数据库在测试期间应映射的数据库的别名。

此设置存在以允许测试多个数据库的主/副本（由某些数据库称为主/从属）配置。有关详细信息，请参阅有关*testing primary/replica configurations*的文档。

NAME

默认值： `None`

运行测试套件时要使用的数据库的名称。

如果默认值（ `None` ）与SQLite数据库引擎一起使用，则测试将使用内存驻留数据库。对于所有其他数据库引擎，测试数据库将使用名称 `'test _' + DATABASE_NAME` 。

请参见*The test database*。

SERIALIZE

New in Django 1.7.1.

布尔值以控制在运行测试之前缺省测试运行器是否将数据库序列化为内存中的JSON字符串（用于在没有事务的情况下在测试之间恢复数据库状态）。如果您没有任何具有*serialized_rollback=True*的测试类，您可以将其设置为 `False` 以加快创建时间。

CREATE_DB

默认值：`True`

这是Oracle特定的设置。

如果设置为 `False` ，则测试表空间将不会在测试开始时自动创建，并在结束时删除。

CREATE_USER

默认值：`True`

这是一个 Oracle-specific 设置.

如果设置为 `False` ，测试用户将不会在测试开始时自动创建，并在最后删除。

USER

默认值：`None`

这是Oracle特定的设置。

连接到将在运行测试时使用的Oracle数据库时使用的用户名。如果没有提供，Django将使用 `'test _' + USER` 。

PASSWORD

默认值：`None`

这是Oracle特定的设置。

连接到运行测试时将使用的Oracle数据库时使用的密码。如果没有提供，Django将使用硬编码的默认值。

TBLSPACE

默认值：`None`

这是Oracle特定的设置。

运行测试时将使用的表空间的名称。如果没有提供，Django将使用 `'test _' + USER` 。

Changed in Django 1.8:

以前，如果未提供，Django使用了 `'test _' + NAME`

TBLSPACE_TMP

默认值：`None`

这是Oracle特定的设置。

运行测试时将使用的临时表空间的名称。如果未提供，Django将使用 `'test _' + USER +` 。

Changed in Django 1.8:

之前Django使用了 `'test _' + NAME + '_ temp'` （如果未提供）。

DATAFILE

New in Django 1.8.

默认值： `None`

这是Oracle特定的设置。

要用于TBLSPACE的数据文件的名称。如果没有提供，Django将使用 `TBLSPACE + '。dbf'` 。

DATAFILE_TMP

New in Django 1.8.

默认值： `None`

这是Oracle特定的设置。

要用于TBLSPACE_TMP的数据文件的名称。如果未提供，Django将使用 `TBLSPACE_TMP + '。dbf'` 。

DATAFILE_MAXSIZE

New in Django 1.8.

默认值： `'500M'`

Changed in Django 1.8:

上一个值为200M，不是用户可自定义的。

这是Oracle特定的设置。

允许DATAFILE增长到的最大大小。

DATAFILE_TMP_MAXSIZE

New in Django 1.8.

默认值： `'500M'`

Changed in Django 1.8:

上一个值为200M，不是用户可自定义的。

这是Oracle特定的设置。

允许DATAFILE_TMP增长到的最大大小。

TEST_CHARSET

自1.7版起已弃用：使用 `TEST` 字典中的 `CHARSET` 条目。

TEST_COLLATION

自1.7版起已弃用：使用 `TEST` 字典中的 `COLLATION` 条目。

TEST_DEPENDENCIES

自1.7版起已弃用：使用 `TEST` 字典中的 `DEPENDENCIES` 条目。

TEST_MIRROR

自1.7版起已弃用：Use the `MIRROR` entry in the `TEST` dictionary.

TEST_NAME

自1.7版起已弃用：使用 `TEST` 字典中的 `NAME` 条目。

TEST_CREATE

自1.7版起已弃用：使用 `TEST` 字典中的 `CREATE_DB` 条目。

TEST_USER

自1.7版起已弃用：使用 `TEST` 字典中的 `USER` 条目。

TEST_USER_CREATE

自1.7版起已弃用：Use the `CREATE_USER` entry in the `TEST` dictionary.

TEST_PASSWD

自1.7版起已弃用：使用 `TEST` 字典中的 `PASSWORD` 条目。

TEST_TBLSPACE

自1.7版起已弃用：Use the `TBLSPACE` entry in the `TEST` dictionary.

TEST_TBLSPACE_TMP

自1.7版起已弃用：Use the `TBLSPACE_TMP` entry in the `TEST` dictionary.

DATABASE_ROUTERS

默认值：`[]`（空列表）

将用于确定在执行数据库查询时要使用哪个数据库的路由器列表。

请参阅有关*automatic database routing in multi database configurations*的文档。

日期格式

Default: `'N j, Y'` (e.g. `Feb. 4, 2003`)

用于在系统任何部分中显示日期字段的默认格式。请注意，如果 `USE_L10N` 设置为 `True`，则区域设置格式具有更高的优先级，将会应用。请参阅 `allowed date format strings`。

另请参阅 `DATETIME_FORMAT`，`TIME_FORMAT` 和 `SHORT_DATE_FORMAT`。

DATE_INPUT_FORMATS

默认：

```
(
    '%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', # '2006-10-25', '10/25/2006', '10/25/06'
    '%b %d %Y', '%b %d, %Y',            # 'Oct 25 2006', 'Oct 25, 2006'
    '%d %b %Y', '%d %b, %Y',            # '25 Oct 2006', '25 Oct, 2006'
    '%B %d %Y', '%B %d, %Y',            # 'October 25 2006', 'October 25, 2006'
    '%d %B %Y', '%d %B, %Y',            # '25 October 2006', '25 October, 2006'
)
```

在日期字段上输入数据时将接受的格式的元组。格式将按顺序尝试，使用第一个有效的格式。请注意，这些格式字符串使用Python的datetime模块语法，而不是来自 `date` Django模板标记的格式字符串。

当 `USE_L10N` 为 `True` 时，区域设置格式具有更高的优先级，将会应用。

另请参阅 `DATETIME_INPUT_FORMATS` 和 `TIME_INPUT_FORMATS`。

DATETIME_FORMAT

Default: `'N j, Y, P'` (e.g. `Feb. 4, 2003, 4 p.m.`)

用于在系统任何部分中显示datetime字段的默认格式。请注意，如果 `USE_L10N` 设置为 `True` ，则区域设置格式具有更高的优先级，将会应用。请参阅 `allowed date format strings` 。

另请参阅 `DATE_FORMAT` ， `TIME_FORMAT` 和 `SHORT_DATETIME_FORMAT` 。

DATETIME_INPUT_FORMATS

默认：

```
(
    '%Y-%m-%d %H:%M:%S',     # '2006-10-25 14:30:59'
    '%Y-%m-%d %H:%M:%S.%f',  # '2006-10-25 14:30:59.000200'
    '%Y-%m-%d %H:%M',        # '2006-10-25 14:30'
    '%Y-%m-%d',              # '2006-10-25'
    '%m/%d/%Y %H:%M:%S',     # '10/25/2006 14:30:59'
    '%m/%d/%Y %H:%M:%S.%f',  # '10/25/2006 14:30:59.000200'
    '%m/%d/%Y %H:%M',        # '10/25/2006 14:30'
    '%m/%d/%Y',              # '10/25/2006'
    '%m/%d/%y %H:%M:%S',     # '10/25/06 14:30:59'
    '%m/%d/%y %H:%M:%S.%f',  # '10/25/06 14:30:59.000200'
    '%m/%d/%y %H:%M',        # '10/25/06 14:30'
    '%m/%d/%y',              # '10/25/06'
)
```

在日期时间字段上输入数据时将接受的格式的元组。格式将按顺序尝试，使用第一个有效的格式。请注意，这些格式字符串使用Python的datetime模块语法，而不是来自 `date` Django模板标记的格式字符串。

当 `USE_L10N` 为 `True` 时，区域设置格式具有更高的优先级，将会应用。

另请参阅 `DATE_INPUT_FORMATS` 和 `TIME_INPUT_FORMATS` 。

DEBUG

默认值： `False`

打开/关闭调试模式的布尔值。

部署网站的时候不要把 `DEBUG` 打开.

你明白了吗？部署网站的时候一定不要把 `DEBUG` 打开.

调试模式的一个重要特性是显示错误页面的细节。当 `DEBUG` 为 `True` 的时候,若你的应用产生了一个异常，Django 会显示追溯细节,包括许多环境变量的元数据, 比如所有当前定义的Django设置(在 `settings.py` 中的).

作为安全措施, Django 将 不会 包括敏感的 (或者可能会被攻击的)设置, 例如 `SECRET_KEY` . 特别是名字中包含下面这些单词的设置:

- `'API'`
- `'KEY'`
- `'PASS'`
- `'SECRET'`
- `'SIGNATURE'`
- `'TOKEN'`

注意，这里使用的是 部分 匹配. `'PASS'` 将匹配 PASSWORD, 另外 `'TOKEN'` 也将匹配 TOKENIZED 等等.

不过，总有一些调试的输出你不希望展现给公众的。文件路径， 配置信息和其他，将会提供信息给攻击者来攻击你的服务器。

另外，很重要的是要记住当你运行时 `DEBUG` 模式打开的话, Django 记住所有执行的 SQL 查询语句。 这在进行 DEBUG 调试时非常有用, 但这会消耗运行服务器的大量内存资源.

最后，如果 `DEBUG` 为 `False` ，你还需要正确设置 `ALLOWED_HOSTS` 。设置错误将导致对所有的请求返回“Bad Request (400)”。

DEBUG_PROPAGATE_EXCEPTIONS

默认值： `False`

如果设置为True，Django的视图函数的正常异常处理将被抑制，异常将向上传播。这对于某些测试设置可能很有用，不应在实际站点上使用。

DECIMAL_SEPARATOR

默认值： `'.'` （点）

格式化十进制数时使用的默认十进制分隔符。

请注意，如果 `USE_L10N` 设置为 `True` ，则区域设置格式具有更高的优先级，将会应用。

另请参阅 `NUMBER_GROUPING` ， `THOUSAND_SEPARATOR` 和 `USE_THOUSAND_SEPARATOR` 。

DEFAULT_CHARSET

默认值：`'utf-8'`

如果未手动指定MIME类型，则用于所有 `HttpResponse` 对象的默认字符集。与 `DEFAULT_CONTENT_TYPE` 配合使用以构造 `Content-Type` 头。

DEFAULT_CONTENT_TYPE

默认值：`'text/html'`

如果未手动指定MIME类型，则用于所有 `HttpResponse` 对象的默认内容类型。与 `DEFAULT_CHARSET` 配合使用以构造 `Content-Type` 头。

DEFAULT_EXCEPTION_REPORTER_FILTER

默认值：`django.views.debug.SafeExceptionReporterFilter`

如果尚未将任何分配给 `HttpRequest` 实例，则使用缺省异常报告器过滤器类。请参阅*Filtering error reports*。

DEFAULT_FILE_STORAGE

默认：`django.core.files.storage.FileSystemStorage`

默认的Storage 类，用于没有指定文件系统的任何和文件相关的操作。参见管理文件。

DEFAULT_FROM_EMAIL

默认值：`'webmaster@localhost'`

用于来自站点管理员的各种自动通信的默认电子邮件地址。这不包括发送到 `ADMINS` 和 `MANAGERS` 的错误消息；有关详细信息，请参阅 `SERVER_EMAIL` 。

DEFAULT_INDEX_TABLESPACE

默认值：`''`（空字符串）

用于不指定一个字段（如果后端支持它）的字段上的索引的默认表空间（请参阅*Tablespaces*）。

DEFAULT_TABLESPACE

默认值：`''`（空字符串）

用于不指定后者的模型（如果后端支持它）的默认表空间（请参阅*Tablespaces*）。

DISALLOWED_USER_AGENTS

默认值： `()` （空元组）

表示不允许访问系统范围内任何页面的User-Agent字符串的编译正则表达式对象列表。用于坏的漫游器/抓取工具。只有在安装 `CommonMiddleware` 时才会使用（请参阅*Middleware*）。

EMAIL_BACKEND

默认： `'django.core.mail.backends.smtp.EmailBackend'`

用于发送邮件的后端。可选的后端参见发送邮件。

EMAIL_FILE_PATH

默认：未指定

`file` 类型的邮件后端保存输出文件时使用的目录。

EMAIL_HOST

默认： `'localhost'`

发送邮件使用的主机。

另见 `EMAIL_PORT` 。

EMAIL_HOST_PASSWORD

默认： `''` （空字符串）

`EMAIL_HOST` 定义的SMTP 服务器使用的密码。这个设置与 `EMAIL_HOST_USER` 一起用于SMTP 服务器的认证。如果两个中有一个为空，Django 则不会尝试认证。

另见 `EMAIL_HOST_USER` 。

EMAIL_HOST_USER

默认： `''` （空字符串）

`EMAIL_HOST` 定义的SMTP 服务器使用的用户名。如果为空，Django 不会尝试认证。

另见 `EMAIL_HOST_PASSWORD` 。

EMAIL_PORT

默认： `25`

`EMAIL_HOST` 定义的SMTP 服务器使用的端口。

EMAIL_SUBJECT_PREFIX

默认值： `'[Django] '`

使用 `django.core.mail.mail_admins` 或 `django.core.mail.mail_managers` 发送的电子邮件的主题行前缀。你可能想要包含结尾空格。

EMAIL_USE_TLS

默认值： `False`

是否使用TLS(安全)当与SMTP服务器的连接。这是用于显式TLS连接,通常在端口587上。如果你正在经历挂连接,看到隐EMAIL_USE_SSL TLS设置。这用于显式TLS连接，通常在端口587上。如果您遇到挂起的连接，请参阅隐式TLS设置 `EMAIL_USE_SSL` 。

EMAIL_USE_SSL

New in Django 1.7.

默认值： `False`

在与SMTP服务器通信时是否使用隐式TLS（安全）连接。在大多数电子邮件文档中，此类型的TLS连接称为SSL。它通常在端口465上使用。如果您遇到问题，请参阅显式TLS设置 `EMAIL_USE_TLS` 。

请注意， `EMAIL_USE_TLS` / `EMAIL_USE_SSL` 是互斥的，因此只能将其中一个设置设置为 `True` 。

EMAIL_SSL_CERTFILE

New in Django 1.8.

默认值： `None`

如果 `EMAIL_USE_SSL` 或 `EMAIL_USE_TLS` 为 `True` ，则可以选择指定要用于SSL连接的PEM格式的证书链文件的路径。

EMAIL_SSL_KEYFILE

New in Django 1.8.

默认值： `None`

如果 `EMAIL_USE_SSL` 或 `EMAIL_USE_TLS` 为 `True` ，您可以选择指定要用于SSL连接的PEM格式的私钥文件的路径。

请注意，设置 `EMAIL_SSL_CERTFILE` 和 `EMAIL_SSL_KEYFILE` 不会导致任何证书检查。它们将传递到底层SSL连接。有关如何处理证书链文件和私钥文件的详细信息，请参阅Python的 `ssl.wrap_socket()` 函数的文档。

EMAIL_TIMEOUT

New in Django 1.8.

默认值： `None`

指定阻止操作（如连接尝试）的超时（以秒为单位）。

FILE_CHARSET

默认值： `'utf-8'`

用于解码从磁盘读取的任何文件的字符编码。这包括模板文件和初始SQL数据文件。

FILE_UPLOAD_HANDLERS

默认：

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler")
```

用于上传的处理程序的元组。更改此设置允许完全自定义 - 甚至替换Django的上传过程。

有关详细信息，请参阅*Managing files*。

FILE_UPLOAD_MAX_MEMORY_SIZE

默认值： `2621440` （即2.5 MB）。

在上传到文件系统之前上传的最大大小（以字节为单位）。有关详细信息，请参阅*Managing files*。

FILE_UPLOAD_DIRECTORY_PERMISSIONS

New in Django 1.7.

默认值： `None`

应用于上传文件过程中创建的目录的数字模式。

此设置还会在使用 `collectstatic` 管理命令时确定收集的静态目录的默认权限。有关覆盖它的详细信息，请参阅 `collectstatic` 。

此值反映了 `FILE_UPLOAD_PERMISSIONS` 设置的功能和注意事项。

FILE_UPLOAD_PERMISSIONS

默认值： `None`

将新上传的文件设置为的数字模式（即 `0o644` ）。有关这些模式意味着什么的更多信息，请参阅 `os.chmod()` 的文档。

如果未给出或 `None` ，您将获得操作系统相关的行为。在大多数平台上，临时文件的模式为 `0o600` ，从内存保存的文件将使用系统的标准umask保存。

出于安全考虑，这些权限不会应用于存储在 `FILE_UPLOAD_TEMP_DIR` 中的临时文件。

此设置还会在使用 `collectstatic` 管理命令时确定收集的静态文件的默认权限。有关覆盖它的详细信息，请参阅 `collectstatic` 。

警告

始终以模式前缀**0.**

如果您不熟悉文件模式，请注意领先的 `0` 非常重要：它表示一个八进制数，这是模式必须指定的方式。如果您尝试使用 `644` ，您将得到完全不正确的行为。

FILE_UPLOAD_TEMP_DIR

默认值： `None`

上传文件时临时存储数据的目录（通常大于 `FILE_UPLOAD_MAX_MEMORY_SIZE` 的文件）。如果 `None` ，Django将使用操作系统的标准临时目录。例如，在* nix风格的操作系统上，这将默认为 `/tmp` 。

有关详细信息，请参阅*Managing files*。

FIRST_DAY_OF_WEEK

默认值： `0` （星期日）

表示一周中第一天的数字。这在显示日历时特别有用。此值仅在不使用格式国际化时或在找不到当前语言环境的格式时使用。

该值必须为0到6之间的整数，其中0表示星期日，1表示星期一，依此类推。

FIXTURE_DIRS

默认值：`()`（空元组）

搜索夹具文件的目录列表，以及搜索顺序中每个应用程序的 `fixtures` 目录。

注意，这些路径应该使用Unix风格的正斜杠，即使在Windows上。

请参阅*Providing initial data with fixtures*和*Fixture loading*。

FORCE_SCRIPT_NAME

默认值: `None`

如果不是 `None` ，这将用作任何HTTP请求中 `SCRIPT_NAME` 环境变量的值。此设置可用于覆盖服务器提供的 `SCRIPT_NAME` 值，该值可能是首选值的重写版本，或者根本不提供。

FORMAT_MODULE_PATH

默认值：`None`

Python包的完整Python路径，其中包含项目语言环境的格式定义。如果不是 `None` ，Django将在名为当前语言环境的目录下检查 `formats.py` 文件，并使用此文件中定义的格式。

例如，如果 `FORMAT_MODULE_PATH` 设置为 `mysite.formats` ，并且当前语言为 `en` （英语），Django将需要一个目录树，

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

Changed in Django 1.8:

您还可以将此设置设置为Python路径列表，例如：

```
FORMAT_MODULE_PATH = [
    'mysite.formats',
    'some_app.formats',
]
```

当Django搜索某种格式时，它将遍历所有给定的Python路径，直到找到一个实际定义给定格式的模块。这意味着在列表中更远的包中定义的格式将优先于在更远的包中的相同格式。

可用的格式有 `DATE_FORMAT` ， `TIME_FORMAT` ， `DATETIME_FORMAT` ， `YEAR_MONTH_FORMAT` ， `MONTH_DAY_FORMAT` ， `SHORT_DATE_FORMAT` ， `SHORT_DATETIME_FORMAT` ， ， `DECIMAL_SEPARATOR` ， `THOUSAND_SEPARATOR` 和 `NUMBER_GROUPING`

IGNORABLE_404_URLS

默认值： `()`

通过电子邮件报告HTTP 404错误时应忽略的编译正则表达式对象列表（请参阅*Error reporting*）。正则表达式与 `request's full paths` （包括查询字符串，如果有）匹配。如果您的网站没有提供通常要求的档案（例如 `favicon.ico` 或 `robots.txt` ），或是遭到脚本小孩的攻击，请使用此方法。

只有在启用 `BrokenLinkEmailsMiddleware` （请参阅*Middleware*）时才会使用此选项。

INSTALLED_APPS

默认值： `()` （空元组）

一个字符串元组，它标明了所有能在django安装的应用每个字符串应该是一个虚线的Python路径：

- 应用程序配置类或
- 包含应用程序的包。

Learn more about application configurations。

Changed in Django 1.7:

`INSTALLED_APPS` 现在支持应用程序配置。

使用应用程序注册表进行自我检查

您的代码不应直接访问 `INSTALLED_APPS` 。请改用 `django.apps.apps` 。

应用程序名称和标签在 `INSTALLED_APPS` 中必须是唯一的

应用程序 `names` - 应用程序包的点分Python路径必须是唯一的。没有办法包含相同的应用程序两次，没有重复其代码在另一个名称下。

应用程序 `标签` - 默认情况下，名称的最后一部分也必须是唯一的。例如，您不能同时包含 `django.contrib.auth` 和 `myproject.auth` 。但是，您可以使用定义不同 `标签` 的自定义配置重新标记应用程序。

无论 `INSTALLED_APPS` 是否引用应用程序包上的应用程序配置类，这些规则都适用。

当多个应用程序提供相同资源（模板，静态文件，管理命令，翻译）的不同版本时， `INSTALLED_APPS` 中首先列出的应用程序优先。

INTERNAL_IPS

默认值： `()` （空元组）

一个IP地址的元组，作为字符串：

- 当 `DEBUG` 为 `True` 时，请参见调试注释
- 如果安装了 `XViewMiddleware` ，请在admindocs中接收X标头（请参阅*The Django admin documentation generator*）

LANGUAGE_CODE

默认值： `'en-us'`

表示此安装的语言代码的字符串。这应该是标准的*language ID format*。例如，美国英语是 `"en-us"` 。另请参阅语言标识符列表和*Internationalization and localization*。

`USE_I18N` 必须处于活动状态才能使此设置生效。

它有两个目的：

- 如果区域中间件未使用，则会决定向所有用户提供哪个翻译。
- 如果区域中间件处于活动状态，则它会提供后备语言，以防用户的首选语言无法确定或网站不支持。当用户的首选语言不存在给定文字的翻译时，它还提供后备翻译。

Changed in Django 1.8:

添加了翻译文字的回退。

有关详细信息，请参见*How Django discovers language preference*。

LANGUAGE_COOKIE_AGE

New in Django 1.7.

默认值： `None` （在浏览器关闭时过期）

语言Cookie的年龄（以秒为单位）。

LANGUAGE_COOKIE_DOMAIN

New in Django 1.7.

默认值： `None`

用于语言Cookie的域。将此字符串设置为跨域Cookie的字符串，例如 `".example.com"` （请注意前导点！），或者对于标准域Cookie使用 `None` 。

在生产站点上更新此设置时要小心。如果您更新此设置以在以前使用标准域Cookie的网站上启用跨网域Cookie，则不会更新具有旧网域的现有用户Cookie。这将导致网站用户无法切换语言，只要这些Cookie持续存在。执行切换的唯一安全可靠的选项是永久更改语言cookie名称（通过 `LANGUAGE_COOKIE_NAME` 设置），并添加一个中间件，将旧值从一个新的cookie复制到一个新的，然后删除旧的。

LANGUAGE_COOKIE_NAME

默认值： `'django_language'`

用于语言Cookie的Cookie的名称。这可以是您想要的（但应该不同于 `SESSION_COOKIE_NAME` ）。请参阅*Internationalization and localization*。

LANGUAGE_COOKIE_PATH

New in Django 1.7.

默认值： `/`

在语言cookie上设置的路径。这应该匹配您的Django安装的URL路径，或者是该路径的父级。

如果您有多个运行在相同主机名下的Django实例，这将非常有用。他们可以使用不同的cookie路径，每个实例只会看到自己的语言cookie。

在生产站点上更新此设置时要小心。如果您将此设置更新为使用比之前使用的更深的路径，则不会更新具有旧路径的现有用户Cookie。这将导致网站用户无法切换语言，只要这些Cookie持续存在。执行切换的唯一安全可靠的选项是永久更改语言Cookie名称（通过 `LANGUAGE_COOKIE_NAME` 设置），并添加中间件，将旧值从旧Cookie复制到新Cookie，然后删除一个。

LANGUAGES

默认：所有可用语言的元组。这个列表不断增长，包括一个副本在这里将不可避免地变得迅速过时。您可以在 `django/conf/global_settings.py` （或查看在线源）中查看当前翻译语言列表。

该列表是格式（*language code*， `语言 名称` ）的二元组的元组 - 例如， `（'ja'， '日语'）` 。这指定了哪些语言可用于语言选择。请参阅*Internationalization and localization*。

一般来说，默认值应该足够了。如果您要将语言选择限制为Django提供的语言的一部分，请仅设置此设置。

如果您定义自定义 `LANGUAGES` 设置，则可以使用 `ugettext_lazy()` 函数将语言名称标记为翻译字符串。

下面是一个示例设置文件：

```
from django.utils.translation import ugettext_lazy as _

LANGUAGES = (
    ('de', _('German')),
    ('en', _('English')),
)
```

LOCALE_PATHS

默认值：`()`（空元组）

Django查找翻译文件的目录的一个元组。请参阅*How Django discovers translations*。

例：

```
LOCALE_PATHS = (
    '/home/www/project/common_files/locale',
    '/var/local/translations/locale',
)
```

Django将在这些路径中查找包含实际翻译文件的 `<locale_code>/LC_MESSAGES` 目录。

LOGGING

默认值：日志配置字典。

包含配置信息的数据结构。此数据结构的内容将作为参数传递到 `LOGGING_CONFIG` 中描述的配置方法。

其中，默认日志配置会在 `DEBUG` 为 `False` 时将HTTP 500服务器错误传递到电子邮件日志处理程序。另请参见*Configuring logging*。

您可以在 `django/utils/log.py` （或查看在线源）中查看默认日志配置。

LOGGING_CONFIG

默认值：`'logging.config.dictConfig'`

将用于在Django项目中配置日志记录的可调用项的路径。默认情况下，Python的dictConfig配置方法实例的点。

如果将 `LOGGING_CONFIG` 设置为 `None` ，将跳过日志配置过程。

Changed in Django 1.7:

以前，默认值为 `'django.utils.log.dictConfig'` 。

MANAGERS

默认值： `()` （空元组）

一个有 `ADMINS` 相同格式的元组， 它指定了谁应该得到 broken link notifications 当 `BrokenLinkEmailsMiddleware` 启用的时候.

MEDIA_ROOT

缺省： `''` （空字符串）

指向存放用户上传文件所在目录的文件系统绝对路径。

例如： `"/var/www/example.com/media/"`

参见 `MEDIA_URL` .

警告：

`MEDIA_ROOT` 和 `STATIC_ROOT` 必须设置为不同的值。在引入（设置） `STATIC_ROOT` 之前, 静态文件的处理将依赖 `MEDIA_ROOT` 。但是，由于这样做会导致产生隐藏的严重安全问题，所以必须进行有效的安全检查以避免这种情况发生。

MEDIA_URL

缺省： `''` (空字符串)

MEDIA*URL*指向 *`MEDIA_ROOT`* 所指定的*media*文件，通过这个地址来*[*管理所存储文件*](../topics/files.html)*。该*URL*设置为非空值时，必须以斜杠*"/"*结束。你需要 *[*配置这些文件用于_](../howto/static-files/index.html#serving-uploaded-files-in-development) 开发环境或线上环境。.

若你打算在模版中使用 `{{ MEDIA_URL }}` , 那么应在 `TEMPLATES` 的 `'context_processors'` 设置中添加 `'django.template.context_processors.media'` .

例如： `"http://media.example.com/"`

警告

如果接受非授信用户上传的内容，将会给系统带来安全风险。关于迁移细节，请参见用户上传内容中安全指南一节。

警告

`MEDIA_URL` 和 `STATIC_URL` 必须设置为不同的值。更多细节，请参见 `MEDIA_ROOT` 。

MIDDLEWARE_CLASSES

默认：

```
('django.middleware.common.CommonMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware')
```

使用中间件类的元组 请参见*Middleware*。

Changed in Django 1.7:

已从此设置中移除 `SessionMiddleware` ， `AuthenticationMiddleware` 和 `MessageMiddleware` 。

MIGRATION_MODULES

默认：

```
{}  # empty dictionary
```

指定可在每个应用程序的基础上找到迁移模块的软件包的字典。此设置的默认值为空字典，但迁移模块的默认软件包名称为 `migrations` 。

例：

```
{'blog': 'blog.db_migrations'}
```

在这种情况下，与 `blog` 应用程序相关的迁移将包含在 `blog.db_migrations` 包中。

如果您提供 `app_label` 参数，则 `makemigrations` 将自动创建软件包（如果尚不存在）。

MONTH_DAY_FORMAT

默认值： `'F j'`

用于Django管理更改列表页面上的日期字段（可能还包括系统的其他部分）的默认格式（仅显示月份和日期时）。

例如，当通过日期明细过滤Django管理更改列表页面时，给定日期的标题显示日期和月份。不同的区域设置具有不同的格式。例如，美国英语会说“1月1日”，而西班牙语可能会说“1 Enero”。

请注意，如果 `USE_L10N` 设置为 `True` ，则相应的区域设置格式具有更高的优先级，并将应用。

请参阅 `allowed date format strings` 。另请参阅 `DATE_FORMAT` ， `DATETIME_FORMAT` ， `TIME_FORMAT` 和 `YEAR_MONTH_FORMAT` 。

NUMBER_GROUPING

默认值： `0`

在数字的整数部分上分组在一起的数字位数。

常用的是显示一千个分隔符。如果此设置为 `0` ，则不会对数字应用分组。如果此设置大于 `0` ，则 `THOUSAND_SEPARATOR` 将用作这些组之间的分隔符。

请注意，如果 `USE_L10N` 设置为 `True` ，则区域设置格式具有更高的优先级，将会应用。

另见 `DECIMAL_SEPARATOR` ， `THOUSAND_SEPARATOR` 和 `USE_THOUSAND_SEPARATOR` 。

PREPEND_WWW

默认值： `False`

是否将“www。”子网域添加到没有网址的网址。仅在安装 `CommonMiddleware` 时使用（请参阅*Middleware*）。另请参阅 `APPEND_SLASH` 。

ROOT_URLCONF

默认值：没有定义

一个字符串，表示根URLconf 的完整Python 导入路径。例如： `"mydjangoapps.urls"` 。每个请求可以覆盖它，方法是设置进来的 `HttpRequest` 对象的 `urlconf` 属性。细节参见*Django* 如何处理一个请求。

SECRET_KEY

默认值： `''` （空字符串）

特定Django安装的密钥。这用于提供*cryptographic signing*，并应设置为唯一的不可预测的值。

`django-admin startproject` 自动向每个新项目添加随机生成的 `SECRET_KEY`

如果未设置 `SECRET_KEY` ，Django将拒绝启动。

警告

将此值保密。

使用已知的 `SECRET_KEY` 运行Django会导致许多Django的安全保护失效，并可能导致特权升级和远程代码执行漏洞。

密钥用于：

- All *sessions* if you are using any other session backend than `django.contrib.sessions.backends.cache` , or if you use `SessionAuthenticationMiddleware` and are using the default `get_session_auth_hash()` .
- 如果您使用 `CookieStorage` 或 `FallbackStorage` ，则所有*messages*。
- `formtools.wizard.views.CookieWizardView` 使用Cookie存储时， `Form wizard`
- 所有 `password_reset()` 令牌。
- 正在进行 `form previews` 。
- 使用*cryptographic signing*的任何用法，除非提供了不同的密钥。

如果您旋转密钥，上述所有内容都将失效。密钥不用于用户的密码，密钥轮换不会影响用户密码。

SECURE_BROWSER_XSS_FILTER

New in Django 1.8.

默认值： `False`

如果 `True` ， `SecurityMiddleware` 设置*X-XSS-Protection: 1; mode=block*标题中所有没有它的响应。

SECURE_CONTENT_TYPE_NOSNIFF

New in Django 1.8.

默认值： `False`

如果 `True` ，则 `SecurityMiddleware` 在尚未拥有它的所有响应上设置*X-Content-Type-Options: nosniff*

SECURE_HSTS_INCLUDE_SUBDOMAINS

New in Django 1.8.

默认值： `False`

如果 `True` ， `SecurityMiddleware` 将 `includeSubDomains` 标记添加到*HTTP Strict Transport Security*除非 `SECURE_HSTS_SECONDS` 设置为非零值，否则它不起作用。

警告

设置不正确可能会不可逆转（一段时间）中断您的网站。首先阅读*HTTP Strict Transport Security*文档。

SECURE_HSTS_SECONDS

New in Django 1.8.

默认值： `0`

如果设置为非零整数值，则 `SecurityMiddleware` 会在尚未拥有它的所有响应上设置*HTTP Strict Transport Security*标题。

警告

设置不正确可能会不可逆转（一段时间）中断您的网站。首先阅读*HTTP Strict Transport Security*文档。

SECURE_PROXY_SSL_HEADER

默认值： `None`

表示表示请求的HTTP头/值组合的元组是安全的。这控制请求对象的 `is_secure()` 方法的行为。

这需要一些解释。默认情况下， `is_secure()` 能够通过查看请求的网址是否使用“https：//”来确定请求是否安全。这对于Django的CSRF保护很重要，可以由您自己的代码或第三方应用程序使用。

如果你的Django应用程序在代理后面，代理可能“吞下”一个请求是HTTPS的事实，使用代理和Django之间的非HTTPS连接。在这种情况下， `is_secure()` 将始终返回 `False` - 即使对于通过HTTPS由最终用户进行的请求。

在这种情况下，您需要配置代理以设置自定义HTTP标头，以通知Django请求是否通过HTTPS传入，并且您希望设置 `SECURE_PROXY_SSL_HEADER` ，以便Django知道哪个报头寻找。

您需要设置一个包含两个元素的元组：要查找的标题的名称和所需的值。例如：

```
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
```

在这里，我们告诉Django我们信任来自我们代理的 `X-Forwarded-Proto` 头部，并且任何时候它的值是 `'https'` 保证是安全的（即，它最初通过HTTPS进来）。显然，如果您控制您的代理或有其他保证它适当地设置/剥离此标头，您应该仅设置此设置。

请注意，标头必须采用 `request.` META - 所有大写，可能从 `HTTP_` 开始。(Remember, Django automatically adds `'HTTP_'` to the start of x-header names before making the header available in `request.` META。)

警告

如果你不知道你在做什么，你可能会在你的网站上打开安全漏洞。 如果你没有设置它，当你应该。认真。

在设置之前，请确保以下所有条件成立（假设上述示例中的值）：

- 你的Django应用程序在代理后面。
- 您的代理会从所有传入的请求中取消 `X-Forwarded-Proto` 标头。换句话说，如果最终用户在其请求中包括该头，代理将丢弃它。
- 您的代理会设置 `X-Forwarded-Proto` 标头，并将其发送到Django，但仅适用于最初通过HTTPS进入的请求。

如果其中任何一个不正确，您应该将此设置设置为 `None` ，并找到另一种确定HTTPS的方法，也许通过自定义中间件。

SECURE_REDIRECT_EXEMPT

New in Django 1.8.

默认值： `[]`

如果网址路径与此列表中的正则表达式匹配，则请求将不会重定向到HTTPS。如果 `SECURE_SSL_REDIRECT` 为 `False` ，则此设置无效。

SECURE_SSL_HOST

New in Django 1.8.

默认值： `None`

如果字符串（例如 `secure.example.com` ），所有SSL重定向将被定向到此主机，而不是原始请求的主机（例如 `www.example.com` ） 。如果 `SECURE_SSL_REDIRECT` 为 `False` ，则此设置无效。

SECURE_ SL_REDIRECT

New in Django 1.8.

默认值： `False` 。

If `True`, the `SecurityMiddleware` *redirects* all non-HTTPS requests to HTTPS (except for those URLs matching a regular expression listed in `SECURE_REDIRECT_EXEMPT`).

注意

如果将此设置为 `True` 会导致无限重定向，则可能意味着您的网站在代理后运行，无法分辨哪些请求是安全的，哪些请求不安全。您的代理可能设置标头来指示安全请求；您可以通过找出该标题并相应地配置 `SECURE_PROXY_SSL_HEADER` 设置来更正该问题。

SERIALIZATION_MODULES

默认值：未定义。

包含序列化程序定义（作为字符串提供）的模块字典，由该序列化类型的字符串标识符键入。例如，要定义YAML序列化程序，请使用：

```
SERIALIZATION_MODULES = {'yaml': 'path.to.yaml_serializer'}
```

SERVER_EMAIL

默认值： `'root@localhost'`

错误消息来源的电子邮件地址，例如发送到 `ADMINS` 和 `MANAGERS` 的电子邮件地址。

为什么我的电子邮件是从其他地址发送的？

此地址仅用于错误消息。不是与 `send_mail()` 一起发送的常规电子邮件的地址来自；有关详情，请参阅 `DEFAULT_FROM_EMAIL` 。

SHORT_DATE_FORMAT

默认值： `m/d/Y` （例如 `12/31/2003` ）

可用于在模板上显示日期字段的可用格式。请注意，如果 `USE_L10N` 设置为 `True` ，则相应的区域设置格式具有更高的优先级，并将应用。请参阅 `allowed date format strings` 。

另请参阅 `DATE_FORMAT` 和 `SHORT_DATETIME_FORMAT` 。

SHORT_DATETIME_FORMAT

预设值： `m / d / Y P` （例如 `12/31/2003 4 pm` ）

可用于在模板上显示datetime字段的可用格式。请注意，如果 `USE_L10N` 设置为 `True` ，则相应的区域设置格式具有更高的优先级，并将应用。请参阅 `allowed date format strings` 。

另请参阅 `DATE_FORMAT` 和 `SHORT_DATE_FORMAT` 。

SIGNING_BACKEND

默认值： `'django.core.signing.TimestampSigner'`

后端用于签名Cookie和其他数据。

另请参阅*Cryptographic signing*文档。

SILENCED_SYSTEM_CHECKS

New in Django 1.7.

默认值： `[]`

您希望永久确认和忽略的系统检查框架生成的消息的标识符列表（即 `["models.W001"]` ）。静音警告将不再输出到控制台；静默错误仍将被打印，但不会阻止管理命令运行。

另请参见*System check framework*文档。

TEMPLATES

New in Django 1.8.

默认:: `[]` （空列表）

Django的模板使用一个列表来进行配置。列表中每一项都是一个字典类型数据，可以配置模板不同的功能。

这里有一个简单的设置，告诉Django模板引擎从已安装的应用程序中的 `模板` 子目录加载模板：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTempla
tes',
        'APP_DIRS': True,
    },
]
```

以下选项适用于所有后端。

BACKEND

默认值：未定义

要使用的模板后端。内置模板后端是：

- `'django.template.backends.django.DjangoTemplates'`
- `'django.template.backends.jinja2.Jinja2'`

通过将 `BACKEND` 设置为完全限定路径（即 `'mypackage.whatever.Backend'` ），您可以使用未随Django提供的模板后端。

NAME

默认值：见下面

此特定模板引擎的别名。它是一个标识符，允许选择引擎进行渲染。别名在所有已配置的模板引擎中必须是唯一的。

它默认为定义引擎类的模块的名称，即 `BACKEND` 的下一段，当没有提供时。例如，如果后端是 `'mypackage.whatever.Backend'` ，则其默认名称为 `'whatever'` 。

DIRS

默认设置： `[]` (空列表)

包含搜索顺序的序列，搜索引擎会按照这个顺序查找template资源文件

APP_DIRS

默认:: `False`

Templates引擎是否应该在已安装的app中查找Template源文件

OPTIONS

默认:: `{}` （空dict）

传递给模板后端的额外参数。可用参数因模板后端而异。

TEMPLATE_CONTEXT_PROCESSORS

默认：

```
("django.contrib.auth.context_processors.auth",
"django.template.context_processors.debug",
"django.template.context_processors.i18n",
"django.template.context_processors.media",
"django.template.context_processors.static",
"django.template.context_processors.tz",
"django.contrib.messages.context_processors.messages")
```

自1.8版起已弃用：在 `DjangoTemplates` 后端的 `OPTIONS` 中设置 `'context_processors'` 选项。

用于填充 `RequestContext` 中上下文的可调用元组。这些callables接受请求对象作为它们的参数，并返回要合并到上下文中的项目的字典。

Changed in Django 1.8:

在Django 1.8中，内置模板上下文处理器已从 `django.core.context_processors` 移至 `django.template.context_processors` 。

TEMPLATE_DEBUG

默认值： `False`

自1.8版起已弃用：在 `DjangoTemplates` 后端的 `OPTIONS` 中设置 `'debug'` 选项。

打开/关闭模板调试模式的布尔值。如果这是 `True` ，花哨的错误页面将显示模板渲染期间引发的任何异常的详细报告。此报告包含模板的相关代码段，并突出显示相应的行。

请注意，如果 `DEBUG` 是 `True` ，Django只会显示精美的错误页面，因此您需要设置它以利用此设置。

另请参见 `DEBUG` 。

TEMPLATE_DIRS

默认值： `()` （空元组）

自1.8版起已弃用：改为设置 `DjangoTemplates` 后端的 `DIRS` 选项。

按照搜索顺序，由 `django.template.loaders.filesystem.Loader` 搜索的模板源文件的位置列表。

注意，这些路径应该使用Unix风格的正斜杠，即使在Windows上。

请参阅 *The Django template language* 。

TEMPLATE_LOADERS

默认：

```
('django.template.loaders.filesystem.Loader',
 'django.template.loaders.app_directories.Loader')
```

自1.8版起已弃用：在 `DjangoTemplates` 后端的 `OPTIONS` 中设置 `'loaders'` 选项。

一个模板加载器类的元组，指定为字符串。每个 `Loader` 类知道如何从特定源导入模板。可选地，可以使用元组而不是字符串。元组中的第一项应该是 `Loader` 的模块，后续项在初始化期间传递到 `Loader` 。请参阅*The Django template language: for Python programmers*。

TEMPLATE_STRING_IF_INVALID

默认值： `''` （空字符串）

自1.8版起已弃用：在 `DjangoTemplates` 后端的 `OPTIONS` 中设置 `'string_if_invalid'` 选项。

输出为字符串，模板系统应该用于无效（例如拼写错误的）变量。请参见*How invalid variables are handled*。

TEST_RUNNER

默认值： `'django.test.runner.DiscoverRunner'`

用于启动测试套件的类的名称。请参见*Using different testing frameworks*。

TEST_NON_SERIALIZED_APPS

New in Django 1.7.

默认值： `[]`

为了在 `TransactionTestCase` 的测试和没有事务的数据库后端之间恢复数据库状态，Django会在开始测试运行时*serialize the contents of all apps with migrations*然后可以在需要它的测试之前从该副本重新加载。

这减慢了测试运行器的启动时间；如果您拥有不需要此功能的应用，则可以在此处添加他们的全名（例如 `'django.contrib.contenttypes'` ），以将其从此序列化过程中排除。

THOUSAND_SEPARATOR

默认值： `,` （逗号）

格式化数字时使用的默认千分位数。此设置仅在 `USE_THOUSAND_SEPARATOR` 为 `True` 且 `NUMBER_GROUPING` 大于 `0` 时使用。

请注意，如果 `USE_L10N` 设置为 `True` ，则区域设置格式具有更高的优先级，将会应用。

另请参阅 `NUMBER_GROUPING` ， `DECIMAL_SEPARATOR` 和 `USE_THOUSAND_SEPARATOR` 。

时间格式

默认值： `'P'` （例如 `4 p.m。` ）

用于在系统任何部分中显示时间字段的默认格式。请注意，如果 `USE_L10N` 设置为 `True` ，则区域设置格式具有更高的优先级，将会应用。请参阅 `allowed date format strings` 。

另请参阅 `DATE_FORMAT` 和 `DATETIME_FORMAT` 。

TIME_INPUT_FORMATS

默认：

```
(
    '%H:%M:%S',     # '14:30:59'
    '%H:%M:%S.%f',  # '14:30:59.000200'
    '%H:%M',        # '14:30'
)
```

在时间字段上输入数据时将接受的格式的元组。格式将按顺序尝试，使用第一个有效的格式。请注意，这些格式字符串使用Python的datetime模块语法，而不是来自 `date` Django模板标记的格式字符串。

当 `USE_L10N` 为 `True` 时，区域设置格式具有更高的优先级，将会应用。

另请参阅 `DATE_INPUT_FORMATS` 和 `DATETIME_INPUT_FORMATS` 。

时区

默认： `'America/Chicago'`

一个字符串或者 `None` ，表示项目的时区。参见时区列表。

注

因为Django 第一次发布时， `TIME_ZONE` 设置为 `'America/Chicago'` ，为了向前兼容，全局设置（在你的项目的 `settings.py` 中没有定义任何内容时使用）仍然保留为 `'America/Chicago'` 。新的项目模板默认为 `'UTC'` 。

注意，它不一定要和服务器的时区一致。例如，一个服务器可上可能有多个Django站点，每个站点都有一个单独的时区设置。

当 `USE_TZ` 为 `False` 时，它将成为Django 存储所有日期和时间时使用的时区。当 `USE_TZ` 为 `True` 时，它是Django 显示模板中以及解释表单中的日期和时间默认使用的时区。

Django 设置 `os.environ['TZ']` 变量为你在 `TIME_ZONE` 设置中指定的时区。所以，你的所有视图和模型都将自动在这个时区中运作。然而，在下面这些情况下，Django 不会设置 `TZ` 环境变量：

- 如果你使用手工配置选项，参见手工配置设置，或
- 如果你指定 `TIME_ZONE = None` 。这将导致Django 使用系统的时区。然而，当 `USE_TZ = True` 时不鼓励这样做，因为这使得本地时间和UTC 之间的转换不太可靠。

如果Django 没有设置 `TZ` 环境变量，那么你需要自己确保你的进程在正确的环境中运行。

注

在Windows 环境中，Django 不能可靠地交替其它时区。如果你在Windows 上运行Django， `TIME_ZONE` 必须设置为与系统时区一致。

USE_ETAGS

默认值： `False`

这是一个布尔变量，它指定是否产生"Etag"头，这种方式会节省带宽但是会降低性能，这个标签在 `CommonMiddleware` (see *Middleware*) 和 `Cache Framework` 中使用(详情见*Django's cache framework*).

USE_I18N

默认值： `True`

这是一个布尔值，它指定Django的翻译系统是否被启用。它提供了一种简单的方式去关闭翻译系统。如果设置为 `False` , Django 会做一些优化，不去加载翻译机制

另请参见 `LANGUAGE_CODE` ， `USE_L10N` 和 `USE_TZ` 。

USE_L10N

默认值： `False`

是一个布尔值，用于决定是否默认进行日期格式本地化。如果此设置为 `True` ，例如Django将使用当前语言环境的格式显示数字和日期。

另请参见 `LANGUAGE_CODE` ， `USE_I18N` 和 `USE_TZ` 。

注意

`django-admin startproject` 创建的默认 `settings.py` 文件包括 `USE_L10N = True` 。

USE_THOUSAND_SEPARATOR

默认值： `False`

一个布尔值，指定是否使用千位分隔符显示数字。当 `USE_L10N` 设置为 `True` ，并且此值也设置为 `True` 时，Django将使用 `THOUSAND_SEPARATOR` 和 `NUMBER_GROUPING` 以格式化数字。

另见 `DECIMAL_SEPARATOR` ， `NUMBER_GROUPING` 和 `THOUSAND_SEPARATOR` 。

USE_TZ

默认: `False`

这是一个布尔值,用来指定是否使用指定的时区(TIME_ZONE)的时间.若为 `True` ,则Django 会使用内建的时区的时间否则, Django 将会使用本地的时间

另请参阅 `TIME_ZONE` ， `USE_I18N` 和 `USE_L10N` 。

注意

使用django-admin startproject创建的项目中的 `settings.py` 文件中,为了方便将 `USE_TZ 设置为 True`

USE_X_FORWARDED_HOST

默认值： `False`

一个布尔值，指定是否使用X-Forwarded-Host头优先于主机头。只有在使用设置此标头的代理时，才应启用此选项。

WSGI_APPLICATION

默认值： `None`

Django的内置服务器（例如 `runserver` ）将使用的WSGI应用程序对象的完整Python路径。The `django-admin startproject` management command will create a simple `wsgi.py` file with an `application` callable in it, and point this setting to that `application` .

如果未设置，将使用 `django.core.wsgi.get_wsgi_application()` 的返回值。在这种情况下， `runserver` 的行为将与以前的Django版本相同。

YEAR_MONTH_FORMAT

默认值： `'F Y'`

用于Django管理更改列表页面上的日期字段（可能还包括系统的其他部分）的默认格式，以便仅显示年份和月份。

例如，当通过日期明细过滤Django管理更改列表页面时，给定月份的标题显示月份和年份。不同的区域设置具有不同的格式。例如，美国英语会说“2006年1月”，而另一个地区可能说“2006 / January”。

请注意，如果 `USE_L10N` 设置为 `True` ，则相应的区域设置格式具有更高的优先级，并将应用。

请参阅 `allowed date format strings` 。另请参阅 `DATE_FORMAT` ， `DATETIME_FORMAT` ， `TIME_FORMAT` 和 `MONTH_DAY_FORMAT` 。

X_FRAME_OPTIONS

默认值： `'SAMEORIGIN'`

`XFrameOptionsMiddleware` 使用的X-Frame-Options标头的默认值。请参阅*clickjacking protection*文档。

Auth

用于 `django.contrib.auth` 的设置。

AUTHENTICATION_BACKENDS

默认： `('django.contrib.auth.backends.ModelBackend',)`

一个元组，包含认证后台的类（字符串形式），用于认证用户。详细信息参见认证后台的文档。

AUTH_USER_MODEL

默认：‘auth.用户'

表示用户的模型。参见自定义用户模型。

警告

在项目的生命周期内（即，一旦完成并迁移了依赖于它的模型），您不能在没有认真努力的情况下更改AUTH*USER_MODEL*设置。它旨在在项目开始时设置，并且其引用的模型必须在其所驻留的应用程序的第一次迁移中可用。有关详细信息，请参阅[*Substituting a custom User model*](../topics/auth/customizing.html#auth-custom-user)。

LOGIN_REDIRECT_URL

默认：`'/accounts/profile/'`

登录之后，`contrib.auth.login` 视图找不到 `next` 参数时，请求被重定向到的URL。

例如，它被 `login_required()` 装饰器使用。

这个设置还接收视图函数名称和*命名的URL 模式*，它可以减少重复的配置，因为这样你就不需要在两个地方定义该URL（`settings` 和URLconf）。

LOGIN_URL

默认：`'/accounts/login/'`

登录的URL，特别是使用 `login_required()` 装饰器的时候。

这个设置还接收视图函数名称和*命名的URL 模式*，它可以减少重复的配置，因为这样你就不需要在两个地方定义该URL（`settings` 和URLconf）。

LOGOUT_URL

默认：`'/accounts/logout/'`

与LOGIN_URL 配对。

PASSWORD_RESET_TIMEOUT_DAYS

默认：`3`

重置密码的链接有效的天数。用于 `django.contrib.auth` 的重置密码功能。

PASSWORD_HASHERS

参见*Django 如何存储密码*。

默认：

```
('django.contrib.auth.hashers.PBKDF2PasswordHasher',
 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
 'django.contrib.auth.hashers.BCryptPasswordHasher',
 'django.contrib.auth.hashers.SHA1PasswordHasher',
 'django.contrib.auth.hashers.MD5PasswordHasher',
 'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
 'django.contrib.auth.hashers.CryptPasswordHasher')
```

消息

`django.contrib.messages` 的设置。

MESSAGE_LEVEL

默认值： `messages.INFO`

设置消息框架将记录的最小消息级别。有关详细信息，请参阅*message levels*。

重要

如果您在设置文件中替换 `MESSAGE_LEVEL` 并依赖于任何内置常量，则必须直接导入常数模块，以避免循环导入的可能性，例如：

```
from django.contrib.messages import constants as message_constan
ts
MESSAGE_LEVEL = message_constants.DEBUG
```

如果需要，可以根据上述*constants table*中的值直接指定常数的数值。

MESSAGE_STORAGE

默认值： `'django.contrib.messages.storage.fallback.FallbackStorage'`

控制Django在哪里存储消息数据。有效值为：

- `'django.contrib.messages.storage.fallback.FallbackStorage'`
- `'django.contrib.messages.storage.session.SessionStorage'`
- `'django.contrib.messages.storage.cookie.CookieStorage'`

有关详细信息，请参阅*message storage backends*。

使用Cookie的后端 - `CookieStorage` 和 `FallbackStorage` - 使用 `SESSION_COOKIE_DOMAIN` ， `SESSION_COOKIE_SECURE` 和 `SESSION_COOKIE_H`
。

MESSAGE_TAGS

默认：

```
{messages.DEBUG: 'debug',
messages.INFO: 'info',
messages.SUCCESS: 'success',
messages.WARNING: 'warning',
messages.ERROR: 'error'}
```

这设置消息级别到消息标记的映射，通常呈现为HTML中的CSS类。如果指定一个值，它将扩展默认值。这意味着您只需要指定需要覆盖的那些值。有关详细信息，请参阅上面的*Displaying messages*。

重要

如果您在设置文件中替换 `MESSAGE_TAGS` 并依赖任何内置常数，则必须直接导入 `constants` 模块，以避免循环导入的可能性，例如：

```
from django.contrib.messages import constants as message_constan
ts
MESSAGE_TAGS = {message_constants.INFO: ''}
```

如果需要，可以根据上述*constants table*中的值直接指定常数的数值。

会话

`django.contrib.sessions` 的设置。

SESSION_CACHE_ALIAS

默认: `默认缓存设置`

使用 缓存存储会话时, 使用何种缓存

SESSION_COOKIE_AGE

默认： `1209600` （2个星期，以秒数为单位）

会话Cookie 的过期时间，以秒数为单位。

SESSION_COOKIE_DOMAIN

默认: `无`

域名用于做会话的cookies.将类似于 `".example.com"` (注意开头的点.) 这样的字符串设置为跨域的cookies, 或者使用 `None` 作为一个标准的域名cookie.

在生产站点上更新此设置时要小心。如果您更新此设置以在之前使用标准域Cookie的网站上启用跨域Cookie，则现有用户Cookie将设置为旧域。这可能会导致他们无法登录，只要这些cookie持续。

此设置还会影响由 `django.contrib.messages` 设置的Cookie。

SESSION_COOKIE_HTTPONLY

默认值： `True`

是否在会话Cookie上使用 `HTTPOnly` 标志。如果设置为 `True` ，客户端JavaScript将无法访问会话Cookie。

HTTPOnly是Set-Cookie HTTP响应标头中包含的标志。它不是Cookie的 **RFC 2109**然而，当它得到尊重时，它可以是一种有用的方式来减轻客户端脚本访问受保护的Cookie数据的风险。

启用它，使攻击者将跨站点脚本漏洞升级为完全劫持用户会话变得不那么简单。有没有什么借口离开这个，或者：如果你的代码依赖于阅读会话cookie从JavaScript，你可能是做错了。

New in Django 1.7.

此设置还会影响由 `django.contrib.messages` 设置的Cookie。

SESSION_COOKIE_NAME

默认值： `'sessionid'`

要用于会话的Cookie的名称。这可以是您想要的（但应该不同于 `LANGUAGE_COOKIE_NAME` ）。

SESSION_COOKIE_PATH

默认值： `'/'`

在会话cookie上设置的路径。这应该匹配您的Django安装的URL路径或该路径的父级。

如果您有多个运行在相同主机名下的Django实例，这将非常有用。他们可以使用不同的cookie路径，每个实例只会看到自己的会话cookie。

SESSION_COOKIE_SECURE

默认值： `False`

是否对会话cookie使用安全cookie。如果此设置为 `True` ，则Cookie将被标记为“安全”，这意味着浏览器可以确保该Cookie仅在HTTPS连接下发送。

由于如果会话cookie未加密发送，数据包嗅探器（例如Firesheep）就劫持用户的会话，这是很平常的，真的没有什么好的借口离开这个。它会阻止你使用不安全的请求会话，这是一件好事。

New in Django 1.7.

此设置还会影响由 `django.contrib.messages` 设置的Cookie。

SESSION_ENGINE

默认： `django.contrib.sessions.backends.db`

控制Django 在哪里存储会话数据。包含的引擎有：

- `'django.contrib.sessions.backends.db'`
- `'django.contrib.sessions.backends.file'`
- `'django.contrib.sessions.backends.cache'`
- `'django.contrib.sessions.backends.cached_db'`
- `'django.contrib.sessions.backends.signed_cookies'`

更多细节参见配置会话引擎。

SESSION_EXPIRE_AT_BROWSER_CLOSE

默认值： `False`

是否在用户关闭浏览器时过期会话。请参阅*Browser-length sessions vs. persistent sessions*。

SESSION_FILE_PATH

默认值： `None`

如果使用基于文件的会话存储，这将设置Django存储会话数据的目录。当使用默认值（ `None` ）时，Django将使用系统的标准临时目录。

SESSION_SAVE_EVERY_REQUEST

默认值： `False`

是否保存每个请求的会话数据。如果这是 `False` （默认值），那么会话数据只有在被修改时才被保存 - 也就是说，如果它的任何字典值被赋值或删除。

SESSION_SERIALIZER

默认值： `'django.contrib.sessions.serializers.JSONSerializer'`

用于序列化会话数据的序列化类的完整导入路径。包括的序列化器有：

- `'django.contrib.sessions.serializers.PickleSerializer'`
- `'django.contrib.sessions.serializers.JSONSerializer'`

有关详细信息，请参阅*Session serialization*，包括使用 `PickleSerializer` 时可能的远程代码执行的警告。

网站

`django.contrib.sites` 的设置。

网站ID

默认：未定义

当前站点在 `django_site` 数据库表中的ID，为一个整数。这是用来让应用程序数据可以连接到特定的网站和一个单一的数据库可以管理多个站点的内容

静态文件

设置为 `django.contrib.staticfiles`.

STATIC_ROOT

默认: `None`

`collectstatic` 用于部署而收集的静态文件的目录的绝对路径。

示例： `"/var/www/example.com/static/"`

如果 *staticfiles* 启用这个服务应用程序 (默认) `collectstatic` 管理命令将收集的静态文件到这个目录查看如何在 *managing static files*有关使用的更多细节。

提醒

这应该是一个 (空目录) 的目录，用于从原始目录收集静态文件到这个目录，便于部署。它 不是永久存储静态文件的地方。你应该在*staticfiles*的 `finders` 找到的目录中，它默认是 `'static/'` app子目录以及您在 `STATICFILES_DIRS` 中包含的任何目录）。

STATIC_URL

默认值: `None`

引用位于 `STATIC_ROOT` 中的静态文件时使用的网址。

示例： `"/static/"` 或 `"http://static.example.com/"`

如果不是 `None` ，则将用作*asset definitions*（ `Media` 类）和*staticfiles app*

如果设置为非空值，它必须以斜杠结尾。

您可能需要*configure these files to be served in development*，并且肯定需要在生产中执行*in production*

STATICFILES_DIRS

默认值： `[]`

此设置定义了在启用 `FileSystemFinder` finder时staticfiles应用程序将遍历的附加位置。如果您使用 `collectstatic` 或 `findstatic` 管理命令或使用静态文件提供视图。

这应该设置为一个列表或元组的字符串，其中包含您的额外文件目录的完整路径例如：

```
STATICFILES_DIRS = (
    "/home/special.polls.com/polls/static",
    "/home/polls.com/polls/static",
    "/opt/webfiles/common",
)
```

请注意，即使在Windows上（例如 `"C:/Users/user/mysite/extra_static_content"` ），这些路径应使用Unix样式的正斜杠。

前缀（可选）

如果要使用其他命名空间引用其中一个位置中的文件，可以（可选）提供前缀 `（前缀， 路径）` 元组，例如：

```
STATICFILES_DIRS = (
    # ...
    ("downloads", "/opt/webfiles/stats"),
)
```

For example, assuming you have `STATIC_URL` set to `'/static/'` , the `collectstatic` management command would collect the “stats” files in a `'downloads'` subdirectory of `STATIC_ROOT` .

这将允许您使用 `'/static/downloads/polls_20101022.tar.gz'` 来引用本地文件 `'/opt/webfiles/stats/polls_20101022.tar.gz'`

```
<a href="{% static "downloads/polls_20101022.tar.gz" %}">
```

STATICFILES_STORAGE

默认值： `'django.contrib.staticfiles.storage.StaticFilesStorage'`

使用 `collectstatic` 管理命令收集静态文件时使用的文件存储引擎。

可在 `django.contrib.staticfiles.storage.staticfiles_storage` 中找到此设置中定义的存储后端的即时使用实例。

有关示例，请参阅*Serving static files from a cloud service or CDN*。

STATICFILES_FINDERS

默认值:

```
("django.contrib.staticfiles.finders.FileSystemFinder",
 "django.contrib.staticfiles.finders.AppDirectoriesFinder")
```

finder后端列表，不同finder用来在不同的位置搜索静态文件。

默认设置是在 `STATICFILES_DIRS` (使用 `django.contrib.staticfiles.finders.FileSystemFinder`)和每个应用的子目录 `static` (使用 `django.contrib.staticfiles.finders.AppDirectoriesFinder`)中搜索.如果存在多个具有相同名称的文件，则将使用找到的第一个文件。

查找器 `django.contrib.staticfiles.finders.DefaultStorageFinder` 默认情况下是被禁用的.如果添加到您的 `STATICFILES_FINDERS` 设置，它将在默认文件存储中查找由 `DEFAULT_FILE_STORAGE` 设置定义的静态文件。

注意

使用 `AppDirectoriesFinder` 查找工具时，请确保您的应用可以通过静态文件找到。只需将应用程序新增至您网站的 `INSTALLED_APPS` 设定即可。

静态文件查找器目前被认为是一个私有接口，因此这个接口是未被文档记录的。

核心设置主题索引

缓存

- 高速缓存
- CACHE_MIDDLEWARE_ALIAS
- CACHE_MIDDLEWARE_KEY_PREFIX
- CACHE_MIDDLEWARE_SECONDS

数据库

- DATABASES
- DATABASE_ROUTERS
- DEFAULT_INDEX_TABLESPACE
- DEFAULT_TABLESPACE

调试

- 调试
- DEBUG_PROPAGATE_EXCEPTIONS

电子邮件

- 管理
- DEFAULT_CHARSET
- DEFAULT_FROM_EMAIL
- EMAIL_BACKEND
- EMAIL_FILE_PATH
- EMAIL_HOST
- EMAIL_HOST_PASSWORD
- EMAIL_HOST_USER
- EMAIL_PORT
- EMAIL_SSL_CERTFILE
- EMAIL_SSL_KEYFILE
- EMAIL_SUBJECT_PREFIX
- EMAIL_TIMEOUT
- EMAIL_USE_TLS
- 经理
- SERVER_EMAIL

报告错误

- DEFAULT_EXCEPTION_REPORTER_FILTER
- IGNORABLE_404_URLS
- 经理
- SILENCED_SYSTEM_CHECKS

文件上传

- DEFAULT_FILE_STORAGE

- FILE_CHARSET
- FILE_UPLOAD_HANDLERS
- FILE_UPLOAD_MAX_MEMORY_SIZE
- FILE_UPLOAD_PERMISSIONS
- FILE_UPLOAD_TEMP_DIR
- MEDIA_ROOT
- MEDIA_URL

全球化（i18n / l10n）

- 日期格式
- DATE_INPUT_FORMATS
- DATETIME_FORMAT
- DATETIME_INPUT_FORMATS
- DECIMAL_SEPARATOR
- FIRST_DAY_OF_WEEK
- FORMAT_MODULE_PATH
- LANGUAGE_CODE
- LANGUAGE_COOKIE_AGE
- LANGUAGE_COOKIE_DOMAIN
- LANGUAGE_COOKIE_NAME
- LANGUAGE_COOKIE_PATH
- 语言
- LOCALE_PATHS
- MONTH_DAY_FORMAT
- NUMBER_GROUPING
- SHORT_DATE_FORMAT
- SHORT_DATETIME_FORMAT
- THOUSAND_SEPARATOR
- 时间格式
- TIME_INPUT_FORMATS
- 时区
- USE_I18N
- USE_L10N
- USE_THOUSAND_SEPARATOR
- USE_TZ
- YEAR_MONTH_FORMAT

HTTP

- DEFAULT_CHARSET
- DEFAULT_CONTENT_TYPE
- DISALLOWED_USER_AGENTS
- FORCE_SCRIPT_NAME
- INTERNAL_IPS
- MIDDLEWARE_CLASSES
- Security

    - {{s.1260}}
    - SECURE_CONTENT_TYPE_NOSNIFF
    - SECURE_HSTS_INCLUDE_SUBDOMAINS
    - SECURE_HSTS_SECONDS
    - SECURE_PROXY_SSL_HEADER
    - SECURE_REDIRECT_EXEMPT
    - SECURE_SSL_HOST
    - SECURE_ SL_REDIRECT
- SIGNING_BACKEND
- USE_ETAGS
- USE_X_FORWARDED_HOST
- WSGI_APPLICATION

记录

- 登录
- LOGGING_CONFIG

楷模

- ABSOLUTE_URL_OVERRIDES
- FIXTURE_DIRS
- INSTALLED_APPS

安全

- Cross Site Request Forgery protection
    - {{s.1278}}
    - CSRF_COOKIE_NAME
    - CSRF_COOKIE_PATH
    - CSRF_COOKIE_SECURE
    - CSRF_FAILURE_VIEW
- SECRET_KEY
- X_FRAME_OPTIONS

序列化

- DEFAULT_CHARSET
- SERIALIZATION_MODULES

模板

- ALLOWED_INCLUDE_ROOTS
- 模板
- TEMPLATE_CONTEXT_PROCESSORS
- TEMPLATE_DEBUG

- TEMPLATE_DIRS
- TEMPLATE_LOADERS
- TEMPLATE_STRING_IF_INVALID

测试

- 数据库：`TEST`
- TEST_NON_SERIALIZED_APPS
- TEST_RUNNER

网址

- APPEND_SLASH
- PREPEND_WWW
- ROOT_URLCONF

应用程序

应用

New in Django 1.7.

Django 包含一个由已安装应用组成的注册表，它保存应用的配置并提供自省。它还维护一个可用的模型的列表。

这个注册表叫做 `apps` ，位于 `django.apps` 中：

```
>>> from django.apps import apps
>>> apps.get_app_config('admin').verbose_name
'Admin'
```

项目和应用

历史上，Django 使用项目 这个词来描述Django 安装的一个应用。现在项目主要通过一个设置模块定义。

应用 这个词表示一个Python 包，它提供某些功能的集合。应用可以在各种项目中重用。

注

最近，这个词有些令人困惑，因为使用词语“Web ?应用”来描述和Django 项目等同的事物渐渐变得常见。

应用包含模型、视图、模板、模板标签、静态文件、URL、中间件等等。它们一般通过 `INSTALLED_APPS` 设置和其它例如URLconf、 `MIDDLEWARE_CLASSES` 设置以及模板继承等机制接入到项目中。

Django 的应用只是一个代码集，它与框架的其它部分进行交互，理解这点很重要。没有类似一个 `应用` 对象这样的东西。然而，有一些地方Django 需要与安装的应用交互，主要用于配置和自省。这是为什么应用注册表要在 `AppConfig` 实例中维持每个安装的应用的元数据。

配置应用

要配置一个应用，请子类化 `AppConfig` 并将这个子类的路径放在 `INSTALLED_APPS` 中。

当 `INSTALLED_APPS` 包含一个应用模块的路径后，Django 将在这个模块中检查一个 `default_app_config` 变量。

如果这个变量有定义，它应该是这个应用的 `AppConfig` 子类的路径。

如果没有 `default_app_config` ，Django 将使用 `AppConfig` 基类。

对于应用的开发者

如果你正在创建一个名叫“Rock ’n’ roll”的可插式应用，下面展示了如何给admin提供一个合适的名字

```
# rock_n_roll/apps.py

from django.apps import AppConfig

class RockNRollConfig(AppConfig):
    name = 'rock_n_roll'
    verbose_name = "Rock ’n’ roll"
```

可以让你的应用像下面一样以默认方式加载这个 `AppConfig` 的子类

```
# rock_n_roll/__init__.py

default_app_config = 'rock_n_roll.apps.RockNRollConfig'
```

这将使得 `INSTALLED_APPS` 只包含 `'rock_n_roll'` 时将使用 `RockNRollConfig` 。这允许你使用 `AppConfig` 功能而不用要求你的用户更新他们的 `INSTALLED_APPS` 设置。

当然，你也可以告诉你的用户将 `'rock_n_roll.apps.RockNRollConfig'` 放在他们的 `INSTALLED_APPS` 设置中。 你甚至可以提供几个具有不同行为的 `AppConfig` 子类，并让使用者通过他们的 `INSTALLED_APPS` 设置选择。

建议的惯例做法是将配置类放在应用的 `apps` 子模块中。但是，Django 不强制这一点。

你必须包含 `name` 属性来让Django 决定该配置适用的应用。你可以定义 `AppConfig` API 参考中记录的任何属性。

注

如果你的代码在应用的 `__init__.py` 中导入应用程序注册表，名称 `apps` 将与 `apps` 子模块发生冲突。最佳做法是将这些代码移到子模块，并将其导入。一种解决方法是以一个不同的名称导入注册表：

```
from django.apps import apps as django_apps
```

对于应用的使用者

如果你在 `anthology` 项目中使用"Rock ’n’ rol"，但您希望显示成"Gypsy jazz"，你可以提供你自己的配置：

```
# anthology/apps.py

from rock_n_roll.apps import RockNRollConfig

class GypsyJazzConfig(RockNRollConfig):
    verbose_name = "Gypsy jazz"

# anthology/settings.py

INSTALLED_APPS = [
    'anthology.apps.GypsyJazzConfig',
    # ...
]
```

再说一次，在 `apps` 子模块定义特定于项目的配置类是一项惯例，并不要求。

应用程序配置

class `AppConfig`

应用程序配置对象存储应用程序的元数据。某些属性可以配置 `AppConfig` 子类中。其他由Django 设置且是只读的。

可配置的属性

`AppConfig.name`

应用的完整Python 路径，例如 `django.contrib.admin` 。

这个属性定义配置适用于哪个应用程序。它必须在所有 `AppConfig` 子类中设置。

它在整个Django 项目中必须是唯一的。

`AppConfig.label`

应用的缩写名称，例如 `'admin'` 。

此属性允许重新标记应用，当两个应用程序有冲突的标签。它默认为 `name` 的最后一部分。它应该是一个有效的 Python 标识符。

它在整个Django 项目中必须是唯一的。

`AppConfig.verbose_name`

应用的适合阅读的名称，例如“Administration”。

此属性默认为 `label.title()` 。

`AppConfig.path`

应用目录的文件系统路径，如 `'/ usr/lib/python3.4/dist-packages/django/contrib/admin'` 。

在大多数情况下，Django 可以自动检测并此设置，但你也可以在 `AppConfig` 子类上提供一个显式的类属性以覆盖它 。在有些情况下是需要这样的；例如，如果应用的包是一个具有多个路径的命名空间包。

只读属性

`AppConfig.module`

应用的根模块，例如 `<module 'django.contrib.admin' from 'django/contrib/admin/__init__` 。

`AppConfig.models_module`

包含模型的模块，例如 `<module 'django.contrib.admin.models' from 'django/contrib/admin/m` 。

如果应用不包含 `models` 模块，它可能为 `None` 。注意与数据库相关的信号，如 `pre_migrate` 和 `post_migrate` 只有在应用具有 `models` 模块时才发出。

方法

`AppConfig.get_models` ()

返回可迭代的 `Model` 类。

`AppConfig.get_model` (*model_name*)

返回具有给定 `model_name` 的 `Model` 。如果没有这种模型存在，抛出 `LookupError` 。 `model_name` 是不区分大小写。

`AppConfig.ready` ()

子类可以重写此方法以执行初始化任务，如注册信号。它在注册表填充完之后立即调用。

你不可以在定义应用程序配置类的模块中导入模型中，但你可以使用 `get_model()` 来通过名称访问模型类，像这样：

```
def ready(self):
    MyModel = self.get_model('MyModel')
```

警告

虽然你可以如上文所述访问模型类，请避免与在你的 `ready()` 实现中与数据库交互。包括执行查询（ `save()` 、 `delete()` 、管理方法等）的模型方法，以及通过 `django.db.connection` 的原始SQL 查询。你的 `ready()` 方法将在每个管理命令启动期间运行。例如，即使测试数据库配置与生产设置是分离的， `manage.py test` 仍会执行一些针对您的 **production** 数据库的查询 ！

注

在常规的初始化过程中， `ready` 方法只由Django 调用一次。但在一些极端情况下，特别是在欺骗安装好的应用的测试， `ready` 可能被调用不止一次。在这种情况下，编写幂等方法，或者在 `AppConfig` 类上放置一个标识来防止应该执行一次的代码重复运行。

Namespace packages as apps (Python 3.3+)

Python versions 3.3 and later support Python packages without an `__init__.py` file. These packages are known as “namespace packages” and may be spread across multiple directories at different locations on `sys.path` (see **PEP 420**).

Django applications require a single base filesystem path where Django (depending on configuration) will search for templates, static assets, etc. Thus, namespace packages may only be Django applications if one of the following is true:

1. The namespace package actually has only a single location (i.e. is not spread across more than one directory.)
2. The `AppConfig` class used to configure the application has a `path` class attribute, which is the absolute directory path Django will use as the single base path for the application.

If neither of these conditions is met, Django will raise `ImproperlyConfigured` .

应用的注册表

`apps`

应用的注册表提供下列公共API。没有在下面列出的方法被认为是私有的，恕不另行通知。

`apps.ready`

布尔属性， 当填充完注册表设置为 `True` 。

`apps.get_app_configs` ()

返回一个由 `AppConfig` 实例组成可迭代对象。

`apps.get_app_config` (*app_label*)

返回具有给定 `app_label` 的 `AppConfig` 。如果没有这种模型存在，抛出 `LookupError` 。

`apps.is_installed` (*app_name*)

检查注册表中是否存在具有给定名称的应用。 `app_name` 是应用的完整名称，例如 `django.contrib.admin` 。

`apps.get_model` (*app_label*, *model_name*)

返回具有给定 `app_label` 和 `model_name` 的 `Model` 。作为快捷方式，此方法还接受 `app_label.model_name` 形式的一个单一参数。 `model_name` 不区分大小写。

如果没有这种模型存在，抛出 `LookupError` 。使用不包含一个点的单个参数调用时将引发 `ValueError` 。

初始化过程

应用是如何载入的

当Django 启动时， `django.setup()` 负责填充应用注册表。

`setup` ()[source]

配置Django：

- 加载设置。
- 设置日志。
- 初始化应用注册表。

自动调用此函数：

- 当运行一个通过Django 的WSGI 支持 的HTTP 服务器。
- 当调用管理命令。

在其他情况下它必须显式调用，例如在普通的 Python 脚本中。

应用注册表初始化分三个阶段。在每个阶段，Django 以 `INSTALLED_APPS` 中的顺序处理所有应用。

1. 首先，Django 会导入 `INSTALLED_APPS` 中的所有应用。

   如果它是一个应用配置类，Django 导入应用的根包，通过其 `name` 属性。如果它是一个Python 包，Django 创建应用的一个默认配置。

   在这个阶段，你的代码不应该将任何模型导入！

换句话说，你的应用程序的根包和定义应用配置类的模块不应该导入任何模型，即使是间接导入。

严格地讲，Django 允许应用配置加载后导入模型。然而，为了避免 `INSTALLED_APPS` 的顺序带来不必要的约束，强烈推荐在这一阶段不导入任何模型。

这一阶段完成后，操作应用配置的API 开始变得可用，例如 `get_app_config()` 。

2. 然后Django 试图导入每个应用的 `models` 子模块，如果有的话。

   你必须在应用的 `models.py` 或 `models/__init__.py` 中定义或导入所有模型。否则，应用注册表在此时可能不会完全填充，这可能导致ORM 出现故障。

   一旦完成该步骤, ? `get_model()` 之类的 model API 可以使用了.

3. 最后，Django 运行每个应用程序配置的 `ready()` 方法。

故障排除

下面是一些在你初始化的时候可能经常碰到的问题:

- `AppRegistryNotReady` 发生这种情况是当导入应用的配置或模型模块时触发取决于应用注册表的代码。

  例如， `ugettext()` 使用应用注册表来查找应用中的翻译目录。若要在导入时翻译，你需要 `ugettext_lazy()` 。（使用 `ugettext()` 将是一个bug，因为翻译会发生在导入的时候，而不是取决于每个请求的语言。）

  模型模块中在导入时使用ORM 执行数据库查询也会引发此异常。ORM 直到所有的模型都可用才能正常运转。

  另一个常见的罪魁祸首，是 `django.contrib.auth.get_user_model()` 。请在导入时使用 `AUTH_USER_MODEL` 设置来引用用户模型。

  如果在一个独立的 Python 脚本中你忘了调用 `django.setup()` ，也会发生此异常。

- `ImportError: cannot import name ...` 如果导入出现循环，则会发生这种情况。

  要消除这种问题，应尽量减少模型模块之间的依赖项，并在导入时尽可能少做工作。为了避免在导入时执行代码，你可以移动它到一个函数和缓存其结果。当你第一次需要其结果时，将执行该代码。这一概念被称为"惰性求值"。

- `django.contrib.admin` 在安装的应用中自动发现 `admin` 。要阻止它，请更改你的 `INSTALLED_APPS` 以包含 `'django.contrib.admin.apps.SimpleAdminConfig'` 而不是 `'django.contrib.admin'` 。

异常

Django异常

DJango会抛出一些它自己的异常，以及Python的标准异常。

Django核心异常

Django核心异常类定义在 `django.core.exceptions` 中。

ObjectDoesNotExist

exception `ObjectDoesNotExist` [source]

`DoesNotExist` 异常的基类；对 `ObjectDoesNotExist` 的 `try/except` 会为所有模型捕获到所有 `DoesNotExist` 异常。

`ObjectDoesNotExist` 和 `DoesNotExist` 的更多信息请见 `get()` 。

FieldDoesNotExist

exception `FieldDoesNotExist` [source]

当被请求的字段在模型或模型的父类中不存在时， `FieldDoesNotExist` 异常由模型的 `_meta.get_field()` 方法抛出。

Changed in Django 1.8:

之前的版本中，异常只在 `django.db.models.fields` 中定义，并不是公共API的一部分。

MultipleObjectsReturned

exception `MultipleObjectsReturned` [source]

`MultipleObjectsReturned` 异常由查询产生，当预期只有一个对象，但是有多个对象返回的时候。这个异常的一个基础版本在 `django.core.exceptions` 中提供。每个模型类都包含一个它的子类版本，它可以用于定义返回多个对象的特定的对象类型。

详见 `get()` 。

SuspiciousOperation

exception `SuspiciousOperation` [source]

当用户进行的操作在安全方面可疑的时候，抛出 `SuspiciousOperation` 异常，例如篡改会话cookie。 `SuspiciousOperation` 的子类包括：

- `DisallowedHost`
- `DisallowedModelAdminLookup`
- `DisallowedModelAdminToField`
- `DisallowedRedirect`
- `InvalidSessionKey`
- `SuspiciousFileOperation`
- `SuspiciousMultipartForm`
- `SuspiciousSession`

如果 `SuspiciousOperation` 异常到达了WSGI处理器层，它会在 `Error` 层记录，并导致 `HttpResponseBadRequest` 异常。 详见日志文档。

PermissionDenied

exception `PermissionDenied` [source]

`PermissionDenied` 异常当用户不被允许来执行请求的操作时产生。

ViewDoesNotExist

exception `ViewDoesNotExist` [source]

当所请求的视图不存在时， `ViewDoesNotExist` 异常由 `django.core.urlresolvers` 产生。

MiddlewareNotUsed

exception `MiddlewareNotUsed` [source]

当中间件没有在服务器配置中出现时，产生 `MiddlewareNotUsed` 异常。

ImproperlyConfigured

exception `ImproperlyConfigured` [source]

DJango配置不当时产生 `ImproperlyConfigured` 异常 -- 例如， `settings.py` 中的值不正确或者不可解析。

FieldError

exception `FieldError` [source]

`FieldError` 异常当模型字段上出现问题时产生。它会由以下原因造成：

- 模型中的字段与抽象基类中相同名称的字段冲突。
- 排序造成了一个死循环。
- 关键词不能由过滤器参数解析。
- 字段不能由查询参数中的关键词决定。
- 连接（join）不能在指定对象上使用。
- 字段名称不可用。
- 查询包含了无效的 order_by参数。

ValidationError

exception `ValidationError` [source]

当表单或模型字段验证失败时抛出 `ValidationError` 异常。关于验证的更多信息，请见表单字段验证, 模型字段验证 和 验证器参考。

NON_FIELD_ERRORS

`NON_FIELD_ERRORS`

在表单或者模型中不属于特定字段的 `ValidationError` 被归类为 `NON_FIELD_ERRORS` 。This constant is used as a key in dictionaries that otherwise map fields to their respective list of errors.

URL解析器异常

URL解析器异常定义在 `django.core.urlresolvers` 中。

Resolver404

exception `Resolver404` [source]

当向 `resolve()` 传递的路径不映射到视图的时候， `Resolver404` 异常由 `django.core.urlresolvers.resolve()` 产生。 它是 `django.http.Http404` 的子类。

NoReverseMatch

exception `NoReverseMatch` [source]

当你的URLconf中的一个匹配的URL不能基于提供的参数识别时， `NoReverseMatch` 异常由 `django.core.urlresolvers` 产生。

Database Exceptions

数据库异常由 `django.db` 导入。

Django封装了标准的数据库异常，以便确保你的DJango代码拥有这些类的通用实现。

exception `Error`

exception `InterfaceError`

exception `DatabaseError`

exception `DataError`

exception `OperationalError`

exception `IntegrityError`

exception `InternalError`

exception `ProgrammingError`

exception `NotSupportedError`

Django数据库异常的包装器的行为和底层的数据库异常一样。详见**PEP 249**，Python 数据库 API 说明 v2.0。

按照 **PEP 3134**， `__cause__` 属性会在原生（底层）的数据库异常中设置，允许访问所提供的任何附加信息。（注意这一属性在Python 2和 3下面都可用，虽然 **PEP 3134**通常只用于Python 3。）

exception `models.``ProtectedError`

使用 `django.db.models.PROTECT` 时，抛出异常来阻止所引用对象的删除。 `models.` ProtectedError is a subclass of `IntegrityError` .

Http异常

HTTP异常由 `django.http` 导入。

UnreadablePostError

exception `UnreadablePostError`

用户取消上传时抛出 `UnreadablePostError` 异常。

事务异常

事务异常定义在 `django.db.transaction` 中。

TransactionManagementError

exception `TransactionManagementError` [source]

对于数据库事务相关的任何问题，抛出 `TransactionManagementError` 异常。

测试框架异常

由DJango `django.test` 包提供的异常。

RedirectCycleError

exception `client.``RedirectCycleError`

New in Django 1.8.

当测试客户端检测到重定向的循环或者过长的链时，抛出 `RedirectCycleError` 异常。

Python异常

Django在适当的时候也会抛出Python的内建异常。进一步的信息请见内建的异常的Python文档。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

django-admin 和 manage.py

django-admin和manage.py

`django-admin` 是用于管理Django的命令行工具集。本文档将概述它的全部功能。

Changed in Django 1.7:

在1.7之前的版本，`django-admin` 以 `django-admin.py` .存在

此外，`manage.py` 会在每个Django项目中自动生成。`manage.py` 是一个对 `django-admin` 的小包装，它可以在交付给 `django-admin` 之前为你做一些事情:

- 他把你的项目包放在python的系统目录中 `sys.path` 。
- 它用于设置 `DJANGO_SETTINGS_MODULE` 环境变量，因此它指向工程的 `settings.py` 文件。
- 通过调用 `django.setup()` 来初始化 Django 的内部变量。

New in Django 1.7:

Django1.7之前的版本没有 `django.setup()`

如果你是通过自带的 `setup.py` 工具来安装Django 的，`django-admin` 脚本应该在你的系统路径里面。如果不在你的系统路径里面，你应该能在你 Python 安装目录下的 `site-packages/django/bin` 里找到，可以创建一个软链接到你的系统路径里面，比如 `/usr/local/bin` 。

对于 Windows 用户，由于不能使用软链接，你可以拷贝 `django-admin.exe` 到你某个系统路径里面，或者是将其添加到系统的 `PATH` 里面 (在 `设置 - 控制面板 - 系统 - 高级系统设置 - 环境变量...`)。

如果你在编写某个项目的时候，通常使用 `manage.py` 要比 `django-admin` 容易一些。如果你需要在不同的 Django 设置文件中来回切换，可以使用 `django-admin` 加上 `DJANGO_SETTINGS_MODULE` 或是 `--settings` 参数。

这篇文档中的控制台操作示例中，一直在使用 `django-admin` ，不过你想用 `manage.py` 也是可以的。

用法

```
$ django-admin <command> [options]
$ manage.py <command> [options]
```

`command` 应是本节文档中所列出的其中一项。`options` ，可选项，应该是可选值中的0项或多项。

获取运行时帮助

`django-admin help`

运行 `django-admin help` 显示使用信息和每个应用的命令列表。

运行 `django-admin help --commands` 显示一个包含所有可用命令的列表

运行 `django-admin help <command>` 来显示某一个命令的描述及其可用的命令列表。

应用程序名称

许多地方都提及"app names"，"app name"是包含models的一个基本package单位。例如，如果你的 `INSTALLED_APPS` 中包含 `'mysite.blog'` ，那么app name就是 `blog` .

确定你使用的版本

`django-admin version`

键入 `django-admin version` 来获取你当前所使用的Django版本。

输出内容的格式在是**PEP 386**:

```
1.4.dev17026
1.4a1
1.4
```

显示调试内容

使用 `--verbosity` 可指定 `django-admin` 应打印到控制台的通知和调试信息量。有关详细信息，请参阅 `--verbosity` 选项的文档。

可用命令

检查<appname ...="" appname=""></appname>

`django-admin check`

Changed in Django 1.7.

使用*system check framework*来检查整个Django项目是否存在常见问题。

系统检查框架将确认您的已安装型号或管理员注册没有任何问题。它还将提供通过将Django升级到新版本引入的常见兼容性问题的警告。其他库和应用程序可能会引入自定义检查。

默认情况下，所有应用都将被选中。您可以通过提供应用标签列表作为参数来检查应用的子集：

```
python manage.py check auth admin myapp
```

如果你没有指定任何一个应用，那么将对全部的应用进行检查。

--tag <tagname>;

*system check framework*执行许多不同类型的检查。Uses the system check framework to inspect the entire Django project for common problems.您可以使用这些标记将执行的检查仅限于特定类别中的检查。例如，要仅执行安全性和兼容性检查，您将运行：

```
python manage.py check --tag security --tag compatibility
```

--list-tags

列出所有可用的标签。

--deploy

New in Django 1.8.

--deploy 选项激活仅与部署设置相关的一些其他检查。

您可以在本地开发环境中使用此选项，但由于本地开发设置模块可能没有很多生产设置，因此您可能希望将 check 命令指向不同的设置模块，通过设置 DJANGO_SETTINGS_MODULE 环境变量，或传递 --settings 选项：

```
python manage.py check --deploy --settings=production_settings
```

或者，您可以直接在生产或分段部署中运行它，以验证正确的设置是否正在使用（省略 --settings ）。您甚至可以将它作为集成测试套件的一部分。

编译消息

django-admin compilemessages

将由 makemessages 创建的.po文件编译为.mo文件以与内置gettext支持一起使用。请参阅*Internationalization and localization*。

使用 --locale 选项（或其较短版本 -l 如果未提供，则处理所有语言环境。

使用 `--exclude` 选项（或其较短版本 `-x` ）指定要从处理中排除的区域设置。如果未提供，则不排除语言环境。

您可以传递 `--use-fuzzy` 选项（或 `-f` ）将模糊翻译包含到编译文件中。

Changed in Django 1.8:

添加了 `--exclude` 和 `--use-fuzzy` 选项。

用法示例：

```
django-admin compilemessages --locale=pt_BR
django-admin compilemessages --locale=pt_BR --locale=fr -f
django-admin compilemessages -l pt_BR
django-admin compilemessages -l pt_BR -l fr --use-fuzzy
django-admin compilemessages --exclude=pt_BR
django-admin compilemessages --exclude=pt_BR --exclude=fr
django-admin compilemessages -x pt_BR
django-admin compilemessages -x pt_BR -x fr
```

createcachetable

`django-admin createcachetable`

创建用于数据库高速缓存后端的高速缓存表。有关详细信息，请参阅*Django's cache framework*。

`--database` 选项可用于指定要安装缓存表的数据库。

Changed in Django 1.7:

不再需要提供高速缓存表名称或 `--database` 选项。Django从您的设置文件中获取此信息。如果已配置多个高速缓存或多个数据库，则将创建所有高速缓存表。

dbshell

`django-admin dbshell`

使用 `USER` ， `PASSWORD` 等中指定的连接参数，为 `ENGINE` 设置中指定的数据库引擎运行命令行客户端。 ，设置。

- 对于PostgreSQL，它运行 `psql` 命令行客户端。
- 对于MySQL，它运行 `mysql` 命令行客户端。
- 对于SQLite，它运行 `sqlite3` 命令行客户端。

此命令假定程序在您的 `PATH` 上，以便简单调用程序名称（ `psql` ， `mysql` ， `sqlite3` 没有办法手动指定程序的位置。

`--database` 选项可用于指定要在其上打开shell的数据库。

diffsettings

`django-admin diffsettings`

显示现在的设置文件和默认的设置文件之间的差异

在默认设置中没有出现的设置后面跟着 `"###"` . 例如，默认设置中没有定义 `ROOT_URLCONF` , 所以在输出 `diffsettings` 时 `ROOT_URLCONF` 后面跟着 `"###"`

可以提供 `--all` 选项以显示所有设置，即使它们具有Django的默认值。此类设置的前缀为 `"###"` 。

dumpdata<app_label ...="" app_label="" app_label.model=""></app_label>

`django-admin dumpdata`

该命令将所有与被命名应用相关联的数据库中的数据输出到标准输出。

如果在dumpdate命令后面未指定Django应用名，则Django项目中安装的所有应用的数据都将被dump到fixture中。

`dumpdata` 命令的输出可作为 `loaddata` 命令的输入

请注意， `dumpdata` 使用模型上的默认管理器来选择要转储的记录。如果您使用*custom manager*作为默认管理器，并过滤一些可用记录，则不会转储所有对象。

可以提供 `--all` 选项来指定 `dumpdata` 应使用Django的基本管理器，转储可能会被自定义管理器过滤或修改的记录。

`--format` `<fmt>`

默认情况下， `dumpdata` 将以JSON格式输出其输出，但您可以使用 `--format` 选项指定另一种格式。目前支持的格式列在*Serialization formats*中。

`--indent` `<num>`

默认情况下， `dumpdata` 将在一行上输出所有数据。这对于人类来说不容易阅读，因此您可以使用 `--indent` 选项来漂亮地打印具有多个缩进空格的输出。

可以提供 `--exclude` 选项以防止特定应用或模型（以 `app_label.` ModelName）。如果将模型名称指定为 `dumpdata` ，则转储的输出将限制为该模型，而不是整个应用程序。您还可以混合应用程序名称和型号名称。

`--database` 选项可用于指定要从中转储数据的数据库。

`--natural-foreign`

New in Django 1.7.

当指定此选项时，Django将使用 `natural_key()` 模型方法将任何外键和多对多关系序列化到定义方法的类型的对象。如果您要转储 `contrib.auth` `Permission` 物件或 `contrib.contenttypes` `ContentType` 物件，旗。有关此和下一个选项的更多详细信息，请参阅*natural keys*文档。

`--natural-primary`

New in Django 1.7.

当指定此选项时，Django不会在此对象的序列化数据中提供主键，因为它可以在反序列化期间计算。

`--natural`

自1.7版起已弃用：等同于 `--natural-foreign` 选项；使用它。

使用*natural keys*来表示任何外键和多对多关系与提供自然键定义的模型。

`--pks`

默认情况下，`dumpdata` 将输出模型的所有记录，但您可以使用 `--pks` 选项指定要过滤的主键的逗号分隔列表。这仅在转储一个模型时可用。

`--output`

New in Django 1.8.

缺省，`dumpdata` 命令会将所有经序列化之后的数据输出到标准输出。使用--output选项允许指定数据被写入的文件。

冲刷

`django-admin flush`

从数据库中删除所有数据，重新执行任何后同步处理程序，并重新安装任何初始数据夹具。

可以提供 `--noinput` 选项以禁止所有用户提示。

`--database` 选项可用于指定要刷新的数据库。

--no-initial-data

使用 `--no-initial-data` 可避免加载initial_data fixture。

inspectdb

`django-admin inspectdb`

内部预测由 `NAME` 设置指向的数据库中的数据库表和视图，并将Django模型模块（`models.py` 文件）输出到标准输出。

如果您有要使用Django的旧数据库，请使用此选项。该脚本将检查数据库并为其中的每个表或视图创建一个模型。

您可能会期望，创建的模型将为表或视图中的每个字段都有一个属性。请注意，`inspectdb` 在其字段名称输出中有一些特殊情况：

- 如果 `inspectdb` 无法将列类型映射到模型字段类型，则会使用 `TextField` 并插入Python注释 `'This在字段旁边的字段 类型 是 a 在生成的模型中。`
- If the database column name is a Python reserved word (such as `'pass'`, `'class'` or `'for'`), `inspectdb` will append `'_field'` to the attribute name. 例如，如果表具有 `'for'` 的列，则生成的模型将具有 `'for_field'` 字段，并设置 `db_column` 属性集到 `'for'`。 `inspectdb` will insert the Python comment `'Field renamed because it was a Python reserved word.'` next to the field.

此功能意味着作为一个快捷方式，而不是作为明确的模型生成。运行它之后，您将需要自己查看生成的模型以进行自定义。特别是，您需要重新排列模型的顺序，以便引用其他模型的模型正确排序。

主键自动对PostgreSQL，MySQL和SQLite进行自动检查，在这种情况下，Django在需要时将 `primary_key=True`。

`inspectdb` 适用于PostgreSQL，MySQL和SQLite。外键检测只适用于PostgreSQL和某些类型的MySQL表。

在模型字段上指定 `default` 时，Django不会创建数据库默认值。类似地，数据库默认值不会转换为模型字段默认值，也不能通过 `inspectdb` 以任何方式检测。

默认情况下，`inspectdb` 创建非托管模型。也就是说，模型的 `Meta` 类中的 `托管 = False` 告诉Django不要管理每个表的创建，修改和删除。如果您希望允许Django管理表的生命周期，您需要将 `managed` 选项更改为 `True`（或者直接删除它，因为 `True`

`--database` 选项可用于指定要内省的数据库。

New in Django 1.8:

添加了检查数据库视图的功能。在以前的版本中，仅检查表（而不是视图）。

loaddata

`django-admin loaddata`

搜索并将命名夹具的内容加载到数据库中。

`--database` 选项可用于指定要将数据加载到的数据库。

`--ignorenonexistent`

`--ignorenonexistent` 选项可用于忽略可能自灯具最初生成后移除的字段和模型。

`--app`

`--app` 选项可用于指定单个应用程序来查找fixture，而不是浏览所有应用程序。

Changed in Django 1.7:

`--app` 已添加。

Changed in Django 1.8:

`--ignorenonexistent` 也忽略不存在的模型。

What's a "fixture"?

*fixture*是包含数据库的序列化内容的文件集合。每个夹具具有唯一的名称，并且包括夹具的文件可以在多个应用中分布在多个目录上。

Django将在三个位置搜索灯具：

1. 在每个安装的应用程序的 `fixtures` 目录中
2. 在 `FIXTURE_DIRS` 设置中命名的任何目录中
3. 在灯具命名的文字路径

Django将加载它找到的任何和所有的灯具在这些位置匹配提供的灯具名称。

如果命名的夹具具有文件扩展名，则只加载该类型的夹具。例如：

```
django-admin loaddata mydata.json
```

将只加载名为 `mydata` 的JSON fixture。fixture扩展名必须与*serializer*（例如， `json` 或 `xml` ）的注册名称相对应。

如果你省略扩展，Django将搜索所有可用的灯具类型匹配的夹具。例如：

```
django-admin loaddata mydata
```

将寻找任何夹具类型称为 `mydata` 。如果fixture目录包含 `mydata.json` ，则该fixture将作为JSON夹具加载。

命名的灯具可以包括目录组件。这些目录将包含在搜索路径中。例如：

```
django-admin loaddata foo/bar/mydata.json
```

将针对每个安装的应用程序搜索 `<app_label>/fixtures/foo/bar/mydata.json` ，`<dirname>/` 在 `FIXTURE_DIRS` 和文本路径 `foo/bar/mydata.json` 中。

当夹具文件被处理时，数据被保存到数据库。不调用模型定义的 `save()` 方法，并且将使用 `raw=True` 调用任何 `pre_save` 或 `post_save` >因为实例只包含模型本地的属性。例如，您可以禁用处理程序访问相关字段，这些字段在夹具加载期间不存在，否则会引发异常：

```
from django.db.models.signals import post_save
from .models import MyModel

def my_handler(**kwargs):
    # disable the handler during fixture loading
    if kwargs['raw']:
        return
    ...

post_save.connect(my_handler, sender=MyModel)
```

你也可以写一个简单的装饰器来封装这个逻辑：

```
from functools import wraps

def disable_for_loaddata(signal_handler):
    """
 Decorator that turns off signal handlers when loading fixture d
ata.
 """
    @wraps(signal_handler)
    def wrapper(*args, **kwargs):
        if kwargs['raw']:
            return
        signal_handler(*args, **kwargs)
    return wrapper

@disable_for_loaddata
def my_handler(**kwargs):
    ...
```

请注意，只要灯具反序列化，这个逻辑将禁用信号，而不仅仅是在 `loaddata` 期间。

注意，夹具文件的处理顺序是未定义的。然而，所有灯具数据被安装为单个事务，因此一个灯具中的数据可以引用另一个灯具中的数据。如果数据库后端支持行级约束，那么将在事务结束时检查这些约束。

`dumpdata` 命令可用于为 `loaddata` 生成输入。

压缩夹具

灯具可以按 `zip` ， `gz` 或 `bz2` 格式压缩。例如：

```
django-admin loaddata mydata.json
```

将查找 `mydata.json` ， `mydata.json.zip` ， `mydata.json.gz` 或 `mydata.json.bz` 。使用包含在压缩压缩归档文件中的第一个文件。

注意，如果发现两个具有相同名称但不同类型的灯具（例如，如果在同一夹具目录中找到 `mydata.json` 和 `mydata.xml.gz` ），灯具安装将中止，并且调用 `loaddata` 中安装的任何数据将从数据库中删除。

MySQL与MyISAM和fixtures

MySQL的MyISAM存储引擎不支持事务或约束，因此如果您使用MyISAM，您将不会获得夹具数据的验证，或者如果找到多个事务文件，则回滚。

数据库特定夹具

如果您在多数据库设置中，您可能需要将夹具数据加载到一个数据库，但不加载到另一个数据库。在这种情况下，您可以添加数据库标识符到您的灯具的名称。

For example, if your `DATABASES` setting has a 'master' database defined, name the fixture `mydata.master.json` or `mydata.master.json.gz` and the fixture will only be loaded when you specify you want to load data into the `master` database.

makemessages

`django-admin makemessages`

运行在当前目录的整个源树上，并拉出所有标记为要翻译的字符串。它在conf / locale（在Django树中）或locale（对于项目和应用程序）目录中创建（或更新）消息文件。更改消息文件后，您需要使用 `compilemessages` 编译它们以与内置gettext支持一起使用。有关详细信息，请参阅*i18n documentation*。

`--all`

使用 `--all` 或 `-a` 选项更新所有可用语言的消息文件。

用法示例：

```
django-admin makemessages --all
```

`--extension`

使用 `--extension` 或 `-e` 选项指定要检查的文件扩展名列表（默认值：“.html”，“.txt”）。

用法示例：

```
django-admin makemessages --locale=de --extension xhtml
```

使用逗号分隔多个扩展程序或多次使用-e或-extension：

```
django-admin makemessages --locale=de --extension=html,txt --ext
ension xml
```

使用 `--locale` 选项（或其较短版本 `-l`

New in Django 1.8.

使用 `--exclude` 选项（或其较短版本 `-x` ）指定要从处理中排除的区域设置。如果未提供，则不排除语言环境。

用法示例：

```
django-admin makemessages --locale=pt_BR
django-admin makemessages --locale=pt_BR --locale=fr
django-admin makemessages -l pt_BR
django-admin makemessages -l pt_BR -l fr
django-admin makemessages --exclude=pt_BR
django-admin makemessages --exclude=pt_BR --exclude=fr
django-admin makemessages -x pt_BR
django-admin makemessages -x pt_BR -x fr
```

Changed in Django 1.7:

在与现有po文件合并时，向 `msgmerge` 命令添加 `--previous` 选项。

`--domain`

使用 `--domain` 或 `-d` 选项更改消息文件的域。目前支持：

- `django` 对于所有 `*.py` ， `*.html` 和 `*.txt`
- `djangojs` 用于 `*.js` 文件

`--symlinks`

在查找新的翻译字符串时，使用 `--symlinks` 或 `-s` 选项跟随符号链接到目录。

用法示例：

```
django-admin makemessages --locale=de --symlinks
```

`--ignore`

使用 `--ignore` 或 `-i` 选项忽略与给定 `glob` 样式模式匹配的文件或目录。使用多次忽略更多。

默认使用这些模式： `'CVS'` ， `'.*'` ， `'*~'` ， `'*.pyc'`

用法示例：

```
django-admin makemessages --locale=en_US --ignore=apps/* --ignore=secret/*.html
```

`--no-default-ignore`

使用 `--no-default-ignore` 选项禁用 `--ignore` 的默认值。

`--no-wrap`

使用 `--no-wrap` 选项禁用在语言文件中将长消息行分成几行。

`--no-location`

使用 `--no-location` 选项不写入 `#： filename：line 文件。` 请注意，使用此选项使技术熟练的翻译人员更难理解每个邮件的上下文。

`--keep-pot`

使用 `--keep-pot` 选项可防止Django在创建.po文件之前删除其生成的临时.pot文件。这对于调试可能阻止创建最终语言文件的错误非常有用。

也可以看看

有关如何自定义 `makemessages` 传递到 `xgettext` 的关键字的说明，请参阅*Customizing the makemessages command*。

makemigrations [<app_label>]</app_label>

`django-admin makemigrations`

New in Django 1.7.

根据检测到的模型改变创建新的迁移。迁移与应用程序之间的关系以及更多内容将在迁移文档中深入介绍。

提供一个或多个应用程序名称作为参数将限制为指定的应用程序创建的迁移以及所需的任何依赖项（例如， `ForeignKey` 的另一端的表）。

`--empty`

`--empty` 选项将导致 `makemigrations` 输出指定应用程序的空迁移，以进行手动编辑。此选项仅适用于高级用户，除非您熟悉迁移格式，迁移操作以及迁移之间的依赖关系，否则不应使用此选项。

`--dry-run`

`--dry-run` 选项显示在不实际将任何迁移文件写入磁盘的情况下将进行的迁移。与 `- verbosity 3` 一起使用此选项还将显示将要写入的完整迁移文件。

`--merge`

`--merge` 选项允许修复迁移冲突。可以提供 `--noinput` 选项以在合并期间抑制用户提示。

`--name``, -n`

New in Django 1.8.

`--name` 选项允许您为迁移指定自定义名称，而不是生成的名称。

`--exit``, -e`

New in Django 1.8.

The `--exit` option will cause `makemigrations` to exit with error code 1 when no migration are created (or would have been created, if combined with `--dry-run`).

migrate [<app_label>[<migrationname>]]</migrationname></app_label>

`django-admin migrate`

New in Django 1.7.

使数据库状态与当前模型集和迁移集同步。迁移与应用程序之间的关系以及更多内容将在迁移文档中深入介绍。

此命令的行为根据提供的参数而有所不同：

- 无参数：所有已迁移的应用程式都会执行所有迁移作业，所有未迁移的应用程式都会与资料库同步处理，
- `<app_label>` ：指定的应用程序运行其迁移，直到最近的迁移。这可能涉及运行其他应用程序的迁移，由于依赖性。
- `&lt； app_label&gt； &lt； migrationname&gt；` ：将数据库模式设置为应用指定迁移的状态，相同的应用程序。如果之前已迁移过指定的迁移，则可能会导致不应用迁移。使用名称 `zero` 取消应用应用程序的所有迁移。

与 `syncdb` 不同，此命令不会提示您创建超级用户（如果不存在）（假设您正在使用 `django.contrib.auth` ）。如果您愿意，可以使用 `createsuperuser` 来执行此操作。

`--database` 选项可用于指定要迁移的数据库。

`--fake`

`--fake` 选项会告诉Django将迁移标记为已应用或未应用，但没有实际运行SQL来更改数据库模式。

这适用于高级用户，如果他们手动应用更改，则直接操作当前迁移状态；请注意，使用 `--fake` 会使迁移状态表进入需要手动恢复的状态，以使迁移正常运行。

New in Django 1.8.

`--fake-initial`

`--fake-initial` 选项可用于允许Django跳过应用程序的初始迁移（如果所有数据库表都具有由所有 `CreateModel` 操作创建的所有模型的名称）迁移已经存在。此选项适用于首次针对预先使用迁移的数据库运行迁移时使用。但是，此选项不会检查匹配的表名以外的匹配数据库模式，因此只有在您确信现有模式与初始迁移中记录的模式匹配时才能安全使用。

自1.8版起已弃用： `--list` 选项已移至 `showmigrations` 命令。

runfcgi [options]

`django-admin runfcgi`

自1.7版起已弃用：FastCGI支持已弃用，将在Django 1.9中删除。

启动一组适用于支持FastCGI协议的任何Web服务器的FastCGI进程。有关详细信息，请参见*FastCGI deployment documentation*。需要来自flup的Python FastCGI模块。

在内部，它包装由 `WSGI_APPLICATION` 设置指定的WSGI应用程序对象。

此命令接受的选项将传递到FastCGI库，不要使用其他Django管理命令通常使用的 `'--'` 前缀。

`protocol`

`protocol=PROTOCOL`

使用协议。*PROTOCOL*可以是 `fcgi` ， `scgi` ， `ajp` 等（默认为 `fcgi` ）

`host`

`host=HOSTNAME`

要监听的主机名。

`port`

`port=PORTNUM`

端口监听。

`socket`

`socket=FILE`

UNIX套接字监听。

`method`

`method=IMPL`

可能的值： `prefork` 或 `threaded` （默认为 `prefork` ）

`maxrequests`

`maxrequests=NUMBER`

在子进程被杀死和新子进程被分叉之前处理的请求数（0表示没有限制）。

`maxspare`

`maxspare=NUMBER`

最大备用进程/线程数。

`minspare`

`minspare=NUMBER`

最小数量的备用进程/线程。

`maxchildren`

`maxchildren=NUMBER`

进程/线程的硬限制数。

`daemonize`

`daemonize=BOOL`

是否从终端分离。

`pidfile`

`pidfile=FILE`

将生成的进程标识写入文件*FILE*。

`workdir`

`workdir=DIRECTORY`

在进行守护进程时切换到目录*DIRECTORY*。

`debug`

`debug=BOOL`

设置为true以启用flup跟踪。

`outlog`

`outlog=FILE`

将stdout写入*FILE*文件。

`errlog`

`errlog=FILE`

将stderr写入*FILE*文件。

`umask`

`umask=UMASK`

在进行守护进程时使用的掩码。该值被解释为八进制数（默认值为 `0o22` ）。

用法示例：

```
django-admin runfcgi socket=/tmp/fcgi.sock method=prefork daemon
ize=true \
    pidfile=/var/run/django-fcgi.pid
```

将FastCGI服务器作为守护程序运行，并将生成的PID写入文件。

runserver [IP地址：端口号]

`django-admin runserver`

启用本地上一个轻量级的Web服务器。默认情况下，服务器运行在IP地址 `127.0.0.1` 的8000端口上。当然，你也可以显式地传递一个IP地址和端口号给它。

如果您以一个普通用户的身份来运行脚本，你可能没有权限在低位端口上运行。低端口数(即1024以下)是预留出来给超级用户（root）的。

这个服务器使用的WSGI application对象是通过在 `WSGI_APPLICATION` .中的设置指定的

不要在生产环境的设置中使用这个服务器。他是没有通过安全审查或者性能测试的。（这是怎么回事。我们所关心的事情是Web框架而不是Web服务器，所以改进这个服务器使它来达到生产环境的性能已经超出了我们的讨论范围。

如果有需要，每当有访问请求时，这一开发服务器便会自动重新载入代码。你并不需要在每次代码有变更后重启服务器来使它生效。然而，一些诸如添加新文件等变更并不会引发服务器的自动重启，所以在这种情况下你仍需手动重启。

Changed in Django 1.7:

当文件完成变更后也会引发开发服务器的重新启动。

如果你使用Linux系统，并且安装了pyinotify，内核信号机制会自动引起开发服务器的重新启动(而不用每秒去轮询文件的最后更改时间戳)。这为大型项目提供了更好的扩展，减少了对代码修改的响应时间，更强大的变化检测和电池使用减少。

New in Django 1.7:

加入对 `pyinotify` 的支持

当你启动服务器之后，在服务器运行过程中每当你的Python代码有变更时，系统的检测框架将会检查整个项目中是否存在一些直观的错误 (参见 `check`). 如果检测到了错误，这些错误信息将会输出至标准输出。

你可以运行多个服务，只要保证它们监听不同的端口。多次执行 `django-admin runserver` 即可

注意默认的IP `127.0.0.1` ，它是不可被网络中的其它主机所访问的。若使开发服务器在网络中被其它主机可见，请使用该主机的IP地址 (如： `192.168.2.1`) 或 `0.0.0.0` ，或者使用 `::` (在IPv6可用的情况下).

You can provide an IPv6 address surrounded by brackets (e.g. `[200a::1]:8000`).你可以使用在括号内使用IPv6地址This will automatically enable IPv6 support.这样会使ipv6地址自动生效

A hostname containing ASCII-only characters can also be used.也可以使用只包含ASCII的主机名.

如果启用了*staticfiles* contrib应用程序（在新项目中为默认），则 `runserver` 命令将被其自己的*runserver*命令覆盖。

如果先前未执行 `migrate` ，则存储迁移历史记录的表在第一次运行 `runserver` 时创建。

`--noreload`

使用 `--noreload` 命令选项可以禁止自动更新这意味着当server开始运行以后，不论你对python代码做了什么修改都不会影响到已经加载到内存中的Python模块.

用法示例：

```
django-admin runserver --noreload
```

`--nothreading`

开发服务器默认开启多线程使用 `--nothreading` 选项可以禁止开发服务器中多线程的使用。

`--ipv6``, -6`

使用 `--ipv6` (or shorter `-6`) 选项告诉Django为开发服务器使用IPV6。这会将默认的IP地址从 `127.0.0.1` 改为 `::1` 。

用法示例：

```
django-admin runserver --ipv6
```

使用不同的端口和地址的示例

端口8000在IP地址 `127.0.0.1` ：

```
django-admin runserver
```

端口8000在IP地址 `1.2.3.4` ：

```
django-admin runserver 1.2.3.4:8000
```

端口7000在IP地址 `127.0.0.1` ：

```
django-admin runserver 7000
```

端口7000在IP地址 `1.2.3.4` ：

```
django-admin runserver 1.2.3.4:7000
```

端口8000在IPv6地址 `::1` ：

```
django-admin runserver -6
```

IPv6地址 `::1` ：

```
django-admin runserver -6 7000
```

端口7000在IPv6地址 `2001:0db8:1234:5678::9` ：

```
django-admin runserver [2001:0db8:1234:5678::9]:7000
```

端口8000在主机的IPv4地址 `localhost` ：

```
django-admin runserver localhost:8000
```

端口8000在主机的IPv6地址 `localhost` ：

```
django-admin runserver -6 localhost:8000
```

使用开发服务器提供静态文件

默认设置中，开发服务器不会为你的网站提供任何的静态文件(比如CSS 文件, images, `MEDIA_URL` 下的文件等等). 如果要配置Django以提供静态媒体，请阅读 *Managing static files (CSS, images)*。

shell

`django-admin shell`

启动Python交互式解释器。

如果安装了任何一个，Django将使用IPython或bpython。如果您安装了一个丰富的shell，但是想强制使用“plain”Python解释器，请使用 `--plain` 选项，如下所示：

```
django-admin shell --plain
```

如果您想要指定IPython或bpython作为您的解释器（如果已安装），您可以使用 `-i` 或 `--interface` 选项指定替代解释器接口：

IPython：

```
django-admin shell -i ipython
django-admin shell --interface ipython
```

bpython：

```
django-admin shell -i bpython
django-admin shell --interface bpython
```

When the “plain” Python interactive interpreter starts (be it because `--plain` was specified or because no other interactive interface is available) it reads the script pointed to by the `PYTHONSTARTUP` environment variable and the `~/.pythonrc.py` script. 如果您不希望此行为，您可以使用 `--no-startup` 选项。例如。：

```
django-admin shell --plain --no-startup
```

showmigrations [<app_label>[<app_label>]]</app_label> </app_label>

`django-admin showmigrations`

New in Django 1.8.

展示项目中所有的迁移

`--list``, -l`

`--list` 选项列出了Django知道的所有应用程序，每个应用程序可用的迁移以及是否应用每个迁移（由 `[X]` 旁边的迁移名称）。

还列出了尚未迁移的应用，但已打印 `（无 迁移）` 。

`--plan``, -p`

`--plan` 选项显示Django将执行的迁移计划以应用迁移。任何提供的应用标签都会被忽略，因为计划可能超出这些应用。与 `--list` 相同，应用的迁移由 `[X]` 标记。对于2以上的冗长，还将显示迁移的所有依赖性。

sql<app_label ...="" app_label=""></app_label>

`django-admin sql`

打印给定应用程序名称的CREATE TABLE SQL语句。

`--database` 选项可用于指定要为其打印SQL的数据库。

sqlall<app_label ...="" app_label=""></app_label>

`django-admin sqlall`

打印给定应用程序名称的CREATE TABLE和初始数据SQL语句。

有关如何指定初始数据的说明，请参阅 `sqlcustom` 的说明。

`--database` 选项可用于指定要为其打印SQL的数据库。

Changed in Django 1.7:

`sql*` 管理命令现在遵守 `DATABASE_ROUTERS` 的 `allow_migrate()` 方法。如果模型已同步到非默认数据库，请使用 `--database` 标志为这些模型获取SQL（以前它们将始终包含在输出中）。

sqlclear<app_label ...="" app_label=""></app_label>

`django-admin sqlclear`

打印给定应用程序名称的DROP TABLE SQL语句。

`--database` 选项可用于指定要为其打印SQL的数据库。

sqlcustom<app_label ...="" app_label=""></app_label>

`django-admin sqlcustom`

打印给定应用程序名称的自定义SQL语句。

对于每个指定应用中的每个模型，此命令查找文件 <app_label>/sql/<modelname>.sql ，其中 <app_label> 应用名称和 <modelname> 是模型的名称小写。例如，如果您有包含 `Story` 模型的应用 `news` ， `sqlcustom` 将尝试读取文件 `news/sql/story.sql` 并将其附加到此命令的输出。

每个SQL文件（如果给出）都应包含有效的SQL。在执行所有模型的表创建语句之后，SQL文件将直接管道传输到数据库中。使用此SQL钩子可以进行任何表修改，或将任何SQL函数插入数据库。

请注意，处理SQL文件的顺序未定义。

`--database` 选项可用于指定要为其打印SQL的数据库。

sqldropindexes<app_label ...="" app_label=""></app_label>

```
django-admin sqldropindexes
```

打印给定应用程序名称的DROP INDEX SQL语句。

`--database` 选项可用于指定要为其打印SQL的数据库。

sqlflush

```
django-admin sqlflush
```

打印将为 `flush` 命令执行的SQL语句。

`--database` 选项可用于指定要为其打印SQL的数据库。

sqlindexes<app_label ...="" app_label=""></app_label>

```
django-admin sqlindexes
```

打印给定应用程序名称的CREATE INDEX SQL语句。

`--database` 选项可用于指定要为其打印SQL的数据库。

sqlmigrate<app_label><migrationname></migrationname></app_label>

```
django-admin sqlmigrate
```

打印命名迁移的SQL。这需要活动的数据库连接，它将用于解析约束名称；这意味着您必须针对要在以后应用它的数据库的副本生成SQL。

请注意， `sqlmigrate` 不会对其输出进行着色。

`--database` 选项可用于指定要为其生成SQL的数据库。

`--backwards`

默认情况下，创建的SQL用于以向前方向运行迁移。传递 `--backwards` 以生成用于取消应用迁移的SQL。

sqlsequencereset<app_label ...="" app_label=""></app_label>

`django-admin sqlsequencereset`

打印用于重置给定应用程序名称的序列的SQL语句。

序列是一些数据库引擎使用的索引，用于跟踪自动递增字段的下一个可用数字。

使用此命令可以生成SQL，这将修复序列与其自动递增的字段数据不同步的情况。

`--database` 选项可用于指定要为其打印SQL的数据库。

squashmigrations<app_label><migration_name></migration_name></app_label>

`django-admin squashmigrations`

将 `app_label` 的迁移（包括 `migration_name` ）迁移到较少的迁移中（如果可能）。由此造成的被挤压的迁移可以安全地与未被挖掘的迁移。有关详细信息，请参阅*Squashing migrations*。

`--no-optimize`

默认情况下，Django将尝试优化迁移中的操作，以减少生成的文件的大小。如果此过程失败或创建不正确的迁移，则传递 `--no-optimize` ，但也请提交有关行为的Django错误报告，因为优化意味着安全。

startapp <app_label>[destination]</app_label>

`django-admin startapp`

在当前目录或给定目标中为给定应用程序名称创建Django应用程序目录结构。

默认情况下，创建的目录包含一个 `models.py` 文件和其他应用程序模板文件。（有关详细信息，请参阅源。）如果只给出应用程序名称，则将在当前工作目录中创建应用程序目录。

如果提供了可选的目标，Django将使用现有目录，而不是创建一个新目录。您可以使用“。”来表示当前工作目录。

例如：

```
django-admin startapp myapp /Users/jezdez/Code/myapp
```

`--template`

使用 `--template` 选项，您可以使用自定义应用程序模板，通过为应用程序模板文件提供目录路径，或者提供压缩文件的路径（ `.tar.gz` ， `.tar.bz2` ， `.tgz` ， `.tbz` ， `.zip` ）模板文件。

例如，在创建 `myapp` 应用程序时，这将在给定目录中查找应用模板：

```
django-admin startapp --template=/Users/jezdez/Code/my_app_templ
ate myapp
```

Django还将使用应用模板文件接受压缩归档的URL（ `http` ， `https` ， `ftp` ），即时下载和提取。

例如，利用Github的功能将仓库公开为zip文件，您可以使用以下URL：

```
django-admin startapp --template=https://github.com/githubuser/d
jango-app-template/archive/master.zip myapp
```

当Django复制应用程序模板文件时，它还通过模板引擎呈现特定文件：扩展名与 `--extension` 选项（默认情况下为 `py` ）匹配的文件，其名称通过 `--name` 选项传递。使用的 `template context`

- 传递给 `startapp` 命令的任何选项（在命令支持的选项之间）
- `app_name` - 传递给命令的应用程序名称
- `app_directory` - 新创建的应用程序的完整路径
- `docs_version` - 文档版本： `'dev'` 或 `'1.x'`

警告

当使用Django模板引擎（默认情况下，所有 `*.py` 文件）呈现应用程序模板文件时，Django也将替换所有包含的模板变量。例如，如果其中一个Python文件包含解释与模板呈现相关的特定功能的docstring，则可能会导致不正确的示例。

要解决此问题，您可以使用 `templatetag` 模板标签来“转义”模板语法的各个部分。

startproject <projectname>[destination]</projectname>

`django-admin startproject`

在当前目录或给定目标中为给定项目名称创建Django项目目录结构。

默认情况下，新目录包含 manage.py 和项目包（包含 settings.py 和其他文件）。有关详细信息，请参阅模板源。

如果仅给出项目名称，则项目目录和项目包将命名为 <projectname> ，并且将在当前工作目录中创建项目目录。

如果提供了可选目标，Django将使用现有目录作为项目目录，并创建 manage.py 和其中的项目包。使用'。'表示当前工作目录。

例如：

```
django-admin startproject myproject /Users/jezdez/Code/myproject
_repo
```

与 startapp 命令一样， --template 选项允许您指定自定义项目模板的目录，文件路径或URL。有关受支持的项目模板格式的详细信息，请参阅 startapp 文档。

例如，在创建 myproject 项目时，将在给定目录中查找项目模板：

```
django-admin startproject --template=/Users/jezdez/Code/my_proje
ct_template myproject
```

Django还将接受使用项目模板文件的压缩归档的URL（ http ， https ， ftp ），即时下载和提取它们。

例如，利用Github的功能将仓库公开为zip文件，您可以使用以下URL：

```
django-admin startproject --template=https://github.com/githubus
er/django-project-template/archive/master.zip myproject
```

当Django复制项目模板文件时，它还通过模板引擎呈现某些文件：扩展名与 --extension 选项（默认情况下为 py ）匹配的文件和文件其名称通过 --name 选项传递。使用的 template context

- 传递到 startproject 命令的任何选项（在命令支持的选项之间）
- project_name - 传递给命令的项目名称
- project_directory - 新创建项目的完整路径
- secret_key - SECRET_KEY 设置的随机密钥
- docs_version - 文档版本： 'dev' 或 '1.x'

另请参阅 startapp 中提及的*rendering warning*。

syncdb

```
django-admin syncdb
```

自1.7版起已弃用：此命令已被弃用，支持 `migrate` 命令，它执行旧的行为以及执行迁移。它现在只是该命令的别名。

`migrate` 的别名。

测试<app identifier="" or="" test=""></app>

`django-admin test`

对所有安装的模型运行测试。有关详细信息，请参阅*Testing in Django*。

`--failfast`

`--failfast` 选项可用于停止运行测试，并在测试失败后立即报告故障。

`--testrunner`

`--testrunner` 选项可用于控制用于执行测试的测试运行器类。如果提供此值，它将覆盖由 `TEST_RUNNER` 设置提供的值。

`--liveserver`

`--liveserver` 选项可用于覆盖运行服务器（与 `LiveServerTestCase` 配合使用）的默认地址。默认值为 `localhost:8081` 。

`--keepdb`

New in Django 1.8.

`--keepdb` 选项可用于在测试运行之间保留测试数据库。这具有跳过创建和销毁操作的优点，这大大减少了运行测试的时间，特别是在大型测试套件中。如果测试数据库不存在，它将在第一次运行时创建，然后保存用于每次后续运行。任何未应用的迁移也将在运行测试套件之前应用于测试数据库。

`--reverse`

New in Django 1.8.

`--reverse` 选项可用于以相反的顺序对测试用例排序。这可能有助于调试未正确隔离且具有副作用的测试。使用此选项时，将保留*Grouping by test class*。

`--debug-sql`

New in Django 1.8.

`--debug-sql` 选项可用于为失败的测试启用*SQL logging*。如果 `--verbosity` 是 `2` ，则也会输出通过测试中的查询。

testserver

`django-admin testserver`

使用给定fixture的数据运行Django开发服务器（如 `runserver` ）。

例如，此命令：

```
django-admin testserver mydata.json
```

...将执行以下步骤：

1. 创建测试数据库，如*The test database*中所述。
2. 使用来自给定夹具的夹具数据填充测试数据库。（有关灯具的更多信息，请参阅上述 `loaddata` 的文档。）
3. 运行Django开发服务器（如 `runserver` ），指向此新创建的测试数据库，而不是生产数据库。

这在许多方面是有用的：

- 当您编写视图如何使用某些灯具数据执行*unit tests*时，您可以使用 `testserver` 手动与Web浏览器中的视图进行交互。
- 假设你正在开发你的Django应用程序，并且有一个你想要交互的数据库的“原始”副本。您可以将数据库转储到fixture（使用 `dumpdata` 命令，如上所述），然后使用 `testserver` 运行带有该数据的Web应用程序。有了这个安排，你有以任何方式混乱你的数据的灵活性，知道你正在做的任何数据更改只是对一个测试数据库。

Note that this server does *not* automatically detect changes to your Python source code (as `runserver` does). 但它会检测对模板的更改。

`--addrport` `[port number or ipaddr:port]`

使用 `--addrport` 指定默认值 `127.0.0.1:8000` 的其他端口或IP地址和端口。此值遵循完全相同的格式，并且具有与 `runserver` 命令的参数完全相同的功能。

例子：

要使用 `fixture1` 和 `fixture2` 在端口7000上运行测试服务器：

```
django-admin testserver --addrport 7000 fixture1 fixture2
django-admin testserver fixture1 fixture2 --addrport 7000
```

（上面的语句是等价的。我们包括它们两者，以证明选项在fixture参数之前或之后是无关紧要的。）

使用 `test` 灯具在1.2.3.4:7000上运行：

```
django-admin testserver --addrport 1.2.3.4:7000 test
```

可以提供 `--noinput` 选项以禁止所有用户提示。

验证

`django-admin validate`

自1.7版起已弃用：由 `check` 命令替换。

验证所有已安装的模型（根据 `INSTALLED_APPS` 设置），并将验证错误打印到标准输出。

应用程序提供的命令

一些命令仅在*implements*的 `django.contrib` 应用程序已启用 `enabled` 时可用。本节按照其应用程序对它们进行分组。

django.contrib.auth

更改密码

`django-admin changepassword`

此命令仅在安装了Django的*authentication system*（ `django.contrib.auth` ）时可用。

允许更改用户的密码。它提示您输入作为参数给定的用户密码的两倍。如果它们都匹配，新密码将立即更改。如果您不提供用户，该命令将尝试更改其用户名与当前用户名匹配的密码。

使用 `--database` 选项指定要为用户查询的数据库。如果没有提供，Django将使用 `default` 数据库。

用法示例：

```
django-admin changepassword ringo
```

createsuperuser

`django-admin createsuperuser`

此命令仅在安装了Django的*authentication system*（ `django.contrib.auth` ）时可用。

创建超级用户帐户（具有所有权限的用户）。如果您需要创建初始超级用户帐户，或者需要以编程方式为您的网站生成超级用户帐户，这将非常有用。

以交互方式运行时，此命令将提示输入新超级用户帐户的密码。当以非交互方式运行时，将不会设置密码，并且超级用户帐户将无法登录，直到为其手动设置密码。

`--username`

`--email`

可以使用命令行上的 `--username` 和 `--email` 参数提供新帐户的用户名和电子邮件地址。如果未提供其中任何一个，则 `createsuperuser` 将在以交互方式运行时提示输入。

使用 `--database` 选项指定将保存超级用户对象的数据库。

New in Django 1.8.

如果要自定义数据输入和验证，可以对管理命令进行子类化，并覆盖 `get_input_data()` 。有关现有实现和方法参数的详细信息，请参阅源代码。例如，如果您在 `REQUIRED_FIELDS` 中有 `ForeignKey` ，并且希望允许创建实例而不是输入现有实例的主键，那么这将非常有用。

django.contrib.gis

ogrinspect

此命令仅在安装*GeoDjango*（ `django.contrib.gis` ）时可用。

请参阅GeoDjango文档中的 `description` 。

django.contrib.sessions

清算

`django-admin clearsessions`

可以作为cron作业运行或直接清除过期的会话。

django.contrib.sitemaps

ping_google

此命令仅在安装了*Sitemaps framework*（ `django.contrib.sitemaps` ）时可用。

请参阅Sitemap说明文件中的 `description` 。

django.contrib.staticfiles

集体

仅当安装了*static files application*（`django.contrib.staticfiles`）时，此命令才可用。

请参阅*staticfiles*文档中的`description`。

findstatic

仅当安装了*static files application*（`django.contrib.staticfiles`）时，此命令才可用。

请参阅*staticfiles*文档中的`description`。

默认选项

虽然一些命令可能允许自己的自定义选项，但每个命令允许以下选项：

`--pythonpath`

用法示例：

```
django-admin migrate --pythonpath='/home/djangoprojects/myproject'
```

将给定的文件系统路径添加到Python 导入搜索路径。如果未提供，`django-admin`将使用`PYTHONPATH`环境变量。

请注意，在`manage.py`中，此选项不必要，因为它会为您设置Python路径。

`--settings`

用法示例：

```
django-admin migrate --settings=mysite.settings
```

显式指定要使用的设置模块。设置模块应该是Python包语法，例如。`mysite.settings`。如果未提供，`django-admin`将使用`DJANGO_SETTINGS_MODULE`环境变量。

请注意，在`manage.py`中，此选项不是必需的，因为默认情况下它会使用当前项目中的`settings.py`。

`--traceback`

用法示例：

```
django-admin migrate --traceback
```

默认情况下，每当发生 `CommandError` 时， `django-admin` 都会显示一个简单的错误消息，但是对于任何其他异常都会显示完整的堆栈跟踪。如果指定 `--traceback` ，则 `CommandError` 引发时， `django-admin` 也将输出完整的堆栈跟踪。

`--verbosity`

用法示例：

```
django-admin migrate --verbosity 2
```

使用 `--verbosity` 可指定 `django-admin` 应打印到控制台的通知和调试信息量。

- `0` 表示无输出。
- `1` 表示正常输出（默认）。
- `2` 表示详细输出。
- `3` 表示非常详细输出。

`--no-color`

New in Django 1.7.

用法示例：

```
django-admin sqlall --no-color
```

默认情况下， `django-admin` 将格式化要着色的输出。例如，错误将以红色打印到控制台，SQL语句将突出显示语法。要防止这种情况并输出纯文本，请在运行命令时传递 `--no-color` 选项。

常用选项

以下选项不适用于每个命令，但是它们对于多个命令是通用的。

`--database`

用于指定命令将在其上操作的数据库。如果未指定，此选项将默认为别名 `default` 。

例如，要使用别名 `master` 从数据库转储数据：

```
django-admin dumpdata --database=master
```

`--exclude`

从输出其内容的应用程序中排除特定应用程序。例如，要从dumpdata的输出中特别排除 `auth` 应用程序，您需要调用：

```
django-admin dumpdata --exclude=auth
```

如果要排除多个应用程序，请使用多个 `--exclude` 指令：

```
django-admin dumpdata --exclude=auth --exclude=contenttypes
```

`--locale`

使用 `--locale` 或 `-l` 如果未提供，则处理所有语言环境。

`--noinput`

使用 `--noinput` 选项来禁止所有用户提示，例如“您确定吗？”确认消息。如果 `django-admin` 作为无人参与的自动脚本执行，这是非常有用的。

额外的美味

语法着色

如果您的终端支持ANSI颜色输出，则 `django-admin` / `manage.py` 命令将使用漂亮的颜色编码输出。如果你将命令的输出传递到另一个程序，它不会使用颜色代码。

在Windows下，本机控制台不支持ANSI转义序列，因此默认情况下没有颜色输出。但是，您可以安装ANSICON第三方工具，Django命令将检测其存在，并将使用其服务的颜色输出，就像在基于Unix的平台上。

用于语法高亮的颜色可以自定义。Django附带三个调色板：

- `dark` ，适用于在黑色背景上显示白色文字的端子。这是默认调色板。
- `light` ，适用于在白色背景上显示黑色文本的终端。
- `nocolor` ，禁用语法高亮显示。

您可以通过设置 `DJANGO_COLORS` 环境变量来指定要使用的调色板来选择调色板。例如，要在Unix或OS / X BASH shell下指定 `light` 选项板，您将在命令提示符下运行以下命令：

```
export DJANGO_COLORS="light"
```

您还可以自定义所使用的颜色。Django指定了使用颜色的多个角色：

- `error` - 主要错误。

- `notice` - 一个小错误。
- `sql_field` - SQL中模型字段的名称。
- `sql_coltype` - SQL中的模型字段的类型。
- `sql_keyword` - 一个SQL关键字。
- `sql_table` - SQL中模型的名称。
- `http_info` - 1XX HTTP信息服务器响应。
- `http_success` - 2XX HTTP成功服务器响应。
- `http_not_modified` - 304 HTTP未修改服务器响应。
- `http_redirect` - 除304之外的3XX HTTP重定向服务器响应。
- `http_not_found` - 404 HTTP未找到服务器响应。
- `http_bad_request` - 除404之外的4XX HTTP错误请求服务器响应。
- `http_server_error` - 5XX HTTP Server错误响应。

可以从以下列表中为这些角色中的每个角色分配特定的前景和背景颜色：

- `black`
- `red`
- `green`
- `yellow`
- `blue`
- `magenta`
- `cyan`
- `white`

然后可以使用以下显示选项修改每种颜色：

- `bold`
- `underscore`
- `blink`
- `reverse`
- `conceal`

颜色规范遵循以下模式之一：

- `role=fg`
- `role=fg/bg`
- `role=fg,option,option`
- `role=fg/bg,option,option`

where `role` is the name of a valid color role, `fg` is the foreground color, `bg` is the background color and each `option` is one of the color modifying options. 然后通过分号分隔多个颜色规范。例如：

```
export DJANGO_COLORS="error=yellow/blue,blink;notice=magenta"
```

将指定使用蓝色闪烁的黄色显示错误，并使用品红色显示通知。所有其他颜色的角色将保持不着色。

也可以通过扩展基本调色板来指定颜色。如果您在颜色规范中放置调色板名称，则该调色板隐含的所有颜色将被加载。所以：

```
export DJANGO_COLORS="light;error=yellow/blue,blink;notice=magenta"
```

将指定使用浅色调调色板中的所有颜色，除了用于将按指定重写的错误和通知的颜色。

New in Django 1.7.

在Django 1.7中添加了通过依赖于ANSICON应用程序在Windows上支持来自 `django-admin` / `manage.py` 实用程序的颜色编码输出。

Bash完成

如果使用Bash shell，请考虑安装Django bash完成脚本，该脚本位于Django发行版中的 `extras/django_bash_completion` 中。它可以启用 `django-admin` 和 `manage.py` 命令的制表符完成，以便您可以，例如...

- 键入 `django-admin` 。
- 按[TAB]查看所有可用选项。
- 键入 `sql` ，然后选择[TAB]，以查看其名称以 `sql` 开头的所有可用选项。

有关如何添加自定义操作，请参阅*Writing custom django-admin commands*。

编写自定义的django-admin命令

应用可以通过 `manage.py` 注册它们自己的动作。例如，你可能想为你正在发布的Django应用添加一个 `manage.py` 动作。在本页文档中，我们将为教程中的 `polls` 应用构建一个自定义的 `closepoll` 命令。

要做到这点，只需向该应用添加一个 `management/commands` 目录。Django将为该目录中名字没有以下划线开始的每个Python模块注册一个 `manage.py` 命令。例如：

```
polls/
    __init__.py
    models.py
    management/
        __init__.py
        commands/
            __init__.py
            _private.py
            closepoll.py
    tests.py
    views.py
```

在Python 2上，请确保 `management` 和 `management/commands` 两个目录都包含 `__init__.py` 文件，否则将检测不到你的命令。

在这个例子中， `closepoll` 命令对任何项目都可使用，只要它们在 `INSTALLED_APPS` 里包含 `polls` 应用。

`_private.py` 将不可以作为一个管理命令使用。

`closepoll.py` 模块只有一个要求 – 它必须定义一个 `Command` 类并扩展自 `BaseCommand` 或其 子类。

> 独立的脚本
>
> 自定义的管理命令主要用于运行独立的脚本或者UNIX crontab和Windows周期任务控制面板周期性执行的脚本。

要实现这个命令，需将 `polls/management/commands/closepoll.py` 编辑成这样：

```
from django.core.management.base import BaseCommand, CommandErro
r
from polls.models import Poll

class Command(BaseCommand):
    help = 'Closes the specified poll for voting'

    def add_arguments(self, parser):
        parser.add_argument('poll_id', nargs='+', type=int)

    def handle(self, *args, **options):
        for poll_id in options['poll_id']:
            try:
                poll = Poll.objects.get(pk=poll_id)
            except Poll.DoesNotExist:
                raise CommandError('Poll "%s" does not exist' %
poll_id)

            poll.opened = False
            poll.save()

            self.stdout.write('Successfully closed poll "%s"' %
poll_id)
```

```
Changed in Django 1.8:

在Django 1.8之前，管理命令基于optparse模块，位置参数传递给*args，可选参
数传递给**options。现在，管理命令使用argparse解析参数，默认所有的参数都传
递给**options，除非你命名你的位置参数为args（兼容模式）。对于新的命令，鼓
励你仅仅使用**options。
```

> 注
>
> 当你使用管理命令并希望提供控制台输出时，你应该写到 `self.stdout` 和 `self.stderr` ，而不能直接打印到 `stdout` 和 `stderr` 。通过使用这些代理方法，测试你自定义的命令将变得非常容易。还请注意，你不需要在消息的末尾加上一个换行符，它将被自动添加，除非你指定 `ending` 参数：
>
> ```
> self.stdout.write("Unterminated line", ending='')
> ```

新的自定义命令可以使用 `python manage.py closepoll <poll_id>` 调用。

`handle()` 接收一个或多个 `poll_ids` 并为他们中的每个设置 `poll.opened` 为 `False` 。如果用户访问任何不存在的 `polls` ，将引发一个 `CommandError` 。 `poll.opened` 属性在教程中并不存在，只是为了这个例子将它添加到 `polls.models.Poll` 中。

接收可选参数

通过接收额外的命令行选项，可以简单地修改 `closepoll` 来删除一个给定的 `poll` 而不是关闭它。这些自定义的选项可以像下面这样添加到 `add_arguments()` 方法中：

```
class Command(BaseCommand):
    def add_arguments(self, parser):
        # Positional arguments
        parser.add_argument('poll_id', nargs='+', type=int)

        # Named (optional) arguments
        parser.add_argument('--delete',
            action='store_true',
            dest='delete',
            default=False,
            help='Delete poll instead of closing it')

    def handle(self, *args, **options):
        # ...
        if options['delete']:
            poll.delete()
        # ...
```

```
Changed in Django 1.8:

之前，只支持标准的optparse库，你必须利用optparse.make_option()扩展命令option_list变量。
```

选项（在我们的例子中为 `delete` ）在 `handle` 方法的 `options` 字典参数中可以访问到。更多关于 `add_argument` 用法的信息，请参考 `argparse` 的Python 文档。

除了可以添加自定义的命令行选项， 管理命令还可以接收一些默认的选项，例如 `--verbosity` 和 `--traceback` 。

管理命令和区域设置

默认情况下， `BaseCommand.execute()` 方法使转换失效，因为某些与Django一起的命令完成的任务要求一个与项目无关的语言字符串（例如，面向用户的内容渲染和数据库填入）。

```
Changed in Django 1.8:

在之前的版本中，Django强制使用"en-us"区域设置而不是使转换失效。
```

如果，出于某些原因，你的自定义的管理命令需要使用一个固定的区域设置，你需要在你的 `handle()` 方法中利用I18N支持代码提供的函数手工地启用和停用它：

```
from django.core.management.base import BaseCommand, CommandErro
r
from django.utils import translation

class Command(BaseCommand):
    ...
    can_import_settings = True

    def handle(self, *args, **options):

        # Activate a fixed locale, e.g. Russian
        translation.activate('ru')

        # Or you can activate the LANGUAGE_CODE # chosen in the
settings:
        from django.conf import settings
        translation.activate(settings.LANGUAGE_CODE)

        # Your command logic here
        ...

        translation.deactivate()
```

另一个需要可能是你的命令只是简单地应该使用设置中设置的区域设置且Django应该保持不让它停用。你可以使用 `BaseCommand.leave_locale_alone` 选项实现这个功能。

虽然上面描述的场景可以工作，但是考虑到系统管理命令对于运行非统一的区域设置通常必须非常小心，所以你可能需要：

- 确保运行命令时 `USE_I18N` 设置永远为 `True` （this is a good example of the potential problems stemming from a dynamic runtime environment that Django commands avoid offhand by deactivating translations）。
- Review the code of your command and the code it calls for behavioral differences when locales are changed and evaluate its impact on predictable behavior of your command.

测试

关于如何测试自定义管理命令的信息可以在测试文档中找到。

Command 对象

`class BaseCommand`

所有管理命令最终继承的基类。

如果你想获得解析命令行参数并在响应中如何调用代码的所有机制，可以使用这个类；如果你不需要改变这个行为，请考虑使用它的子类。

继承 `BaseCommand` 类要求你实现 `handle()` 方法。

属性

所有的属性都可以在你派生的类中设置，并在 `BaseCommand` 的子类中使用。

`BaseCommand.args`

一个字符串，列出命令接收的参数，适合用于帮助信息；例如，接收一个应用名称列表的命令可以设置它为‘ `<app_label app_label ...>` ’。

```
Deprecated since version 1.8:

现在，应该在add_arguments()方法中完成，通过调用parser.add_argument()方法。参见上面的closepoll例子。
```

`BaseCommand.can_import_settings`

一个布尔值，指示该命令是否需要导入Django的设置的能力；如果为 `True` ， `execute()` 将在继续之前验证这是否可能。默认值为 `True` 。

`BaseCommand.help`

命令的简短描述，当用户运行 `python manage.py help <command>` 命令时将在帮助信息中打印出来。

`BaseCommand.missing_args_message`

```
New in Django 1.8.
```

如果你的命令定义了必需的位置参数，你可以自定义参数缺失时返回的错误信息。默认是由 `argparse` 输出的 (“too few arguments”)。

`BaseCommand.option_list`

这是optparse选项列表，将赋值给命令的OptionParser用于解析命令。

```
Deprecated since version 1.8:

现在，你应该覆盖`add_arguments()`方法来添加命令行接收的自定义参数。参见上面的例子。
```

`BaseCommand.output_transaction`

一个布尔值，指示命令是否输出SQL语句；如果为 `True` ，输出将被自动用 `BEGIN;` 和 `COMMIT;` 封装。默认为 `False` 。

`BaseCommand.requires_system_checks`

> New in Django 1.7.

一个布尔值；如果为 `True` ，在执行该命令之前将检查整个Django项目是否有潜在的问题。如果 `requires_system_checks` 缺失，则使用 `requires_model_validation` 的值。如果后者的值也缺失，则使用默认值（ `True` ）。同时定义 `requires_system_checks` 和 `requires_model_validation` 将导致错误。

`BaseCommand.requires_model_validation`

> Deprecated since version 1.7:
>
> 被requires_system_checks代替

一个布尔值；如果为 `True` ，将在执行命令之前作安装的模型的验证。默认为 `True` 。若要验证一个单独应用的模型而不是全部应用的模型，可以调用在 `handle()` 中调用 `validate()` 。

`BaseCommand.leave_locale_alone`

一个布尔值，指示设置中的区域设置在执行命令过程中是否应该保持而不是强制设成'en-us'。

默认值为 `False` 。

如果你决定在你自定义的命令中修改该选项的值，请确保你知道你正在做什么。 如果它创建对区域设置敏感的数据库内容，这种内容不应该包含任何转换（比如 `django.contrib.auth` 权限发生的情况），因为将区域设置变成与实际上默认的'en-us' 不同可能导致意外的效果。更进一步的细节参见上面的管理命令和区域设置一节。

当 `can_import_settings` 选项设置为 `False` 时，该选项不可以也为 `False` ，因为尝试设置区域设置需要访问 `settings` 。这种情况将产生一个 `CommandError` 。

方法

`BaseCommand` 有几个方法可以被覆盖，但是只有 `handle()` 是必须实现的。

> 在子类中实现构造函数
>
> 如果你在 `BaseCommand的` 子类中实现 `__init__` ，你必须调用 `BaseCommand` 的 `__init__` ：
>
> ```
> class Command(BaseCommand):
>     def __init__(self, *args, **kwargs):
>         super(Command, self).__init__(*args, **kwargs)
>         # ...
> ```

`BaseCommand.add_arguments(parser)`

```
New in Django 1.8.
```

添加解析器参数的入口，以处理传递给命令的命令行参数。自定义的命令应该覆盖这个方法以添加命令行接收的位置参数和可选参数。当直接继承 `BaseCommand` 时不需要调用 `super()` 。

`BaseCommand.get_version()`

返回Django的版本，对于所有内建的Django命令应该都是正确的。用户提供的命令可以覆盖这个方法以返回它们自己的版本。

`BaseCommand.execute(*args, **options)`

执行这个命令，如果需要则作系统检查（通过 `requires_system_checks` 属性控制）。如果该命令引发一个 `CommandError` ，它将被截断并打印到标准错误输出。

> 在你的代码中调用管理命令
>
> 不应该在你的代码中直接调用 `execute()` 来执行一个命令。请使用 `call_command` 。

`BaseCommand.handle(*args, **options)`

命令的真正逻辑。子类必须实现这个方法。

`BaseCommand.check(app_configs=None, tags=None, display_num_errors=Fa`

```
New in Django 1.7.
```

利用系统的检测框架检测全部Django项目的潜在问题。严重的问题将引发 `CommandError` ；警告会输出到标准错误输出；次要的通知会输出到标准输出。

如果 `app_configs` 和 `tags` 都为 `None` ，将进行所有的系统检查。 `tags` 可以是一个要检查的标签列表，比如 `compatibility` 或 `models` 。

`BaseCommand.validate(app=None, display_num_errors=False)`

> Deprecated since version 1.7:
>
> 被check命令代替

如果 `app` 为 `None` ，那么将检查安装的所有应用的错误。

BaseCommand 的子类

`class AppCommand`

这个管理命令接收一个或多个安装的应用标签作为参数，并对它们每一个都做一些动作。

子类不用实现 `handle()` ，但必须实现 `handle_app_config()` ，它将会为每个应用调用一次。

`AppCommand.handle_app_config(app_config, **options)`

对 `app_config` 完成命令行的动作，其中 `app_config` 是 `AppConfig` 的实例，对应于在命令行上给出的应用标签。

> Changed in Django 1.7:
>
> 以前，AppCommand子类必须实现 `handle_app(app, **options)` ，其中 `app` 是一个模型模块。新的API可以不需要模型模块来处理应用。迁移的最快的方法如下：
>
> ```
> def handle_app_config(app_config, **options):
>     if app_config.models_module is None:
>         return                                  # Or raise a
> n exception.
>     app = app_config.models_module
>     # Copy the implementation of handle_app(app_config, **op
> tions) here.
> ```
>
> 然而，你可以通过直接使用 `app_config` 的属性来简化实现。

`class LabelCommand`

这个管理命令接收命令行上的一个或多个参数（标签），并对它们每一个都做一些动作。

子类不用实现 `handle()` ，但必须实现 `handle_label()` ，它将会为每个标签调用一次。

`LabelCommand.handle_label(label, **options)`

对 `label` 完成命令行的动作， `label` 是命令行给出的字符串。

`class NoArgsCommand`

> Deprecated since version 1.8:
>
> 使用BaseCommand代替，它默认也不需要参数。

这个命令不接收命令行上的参数。

子类不需要实现 `handle()` ，但必须实现 `handle_noargs()` ； `handle()` 本身已经被覆盖以保证不会有参数传递给命令。

`NoArgsCommand.handle_noargs(**options)`

完成这个命令的动作

Command 的异常

`class CommandError`

异常类，表示执行一个管理命令时出现问题。

如果这个异常是在执行一个来自命令行控制台的管理命令时引发，它将被捕获并转换成一个友好的错误信息到合适的输出流（例如，标准错误输出）；因此，引发这个异常（并带有一个合理的错误描述）是首选的方式来指示在执行一个命令时某些东西出现错误。

如果管理命令从代码中通过call_command调用，那么需要时捕获这个异常由你决定。

> 译者：Django 文档协作翻译小组，原文：Adding custom commands。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

测试

Django中的测试

自动化测试对于现代web开发者来说，是非常实用的除错工具。你可以使用一系列测试-- 测试套件 -- 来解决或者避免大量问题：

- 当你编写新代码的时候，你可以使用测试来验证你的代码是否像预期一样工作。
- 当你重构或者修改旧代码的时候，你可以使用测试来确保你的修改不会在意料之外影响到你的应用的应为。

测试web应用是个复杂的任务，因为web应用由很多的逻辑层组成 -- 从HTTP层面的请求处理，到表单验证和处理，到模板渲染。使用Django的测试执行框架和各种各样的工具，你可以模拟请求，插入测试数据，检查你的应用的输出，以及大体上检查你的代码是否做了它应该做的事情。

最好的一点是，它非常简单。

在Django中编写测试的最佳方法是，使用构建于Python标准库的unittest模块。这在编写和运行测试 文档中会详细介绍。

你也可以使用任何其它 Python 的测试框架；Django为整合它们提供了API和工具。这在高级测试话题的使用不同的测试框架 一节中描述。

- 编写和运行测试
- 测试工具
- 高级测试话题

> 译者：Django 文档协作翻译小组，原文：Introduction。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写并运行测试用例

参考

testing tutorial，*testing tools reference*和*advanced testing topics*。

本文档分为2个主要单元。首先，我们讲解如何利用Django编写测试.之后，我们讲解如何运行测试。

编写测试

Django的单元测试使用的是Python标准库：`unittest`。该模块是采用基于类的测试。

unittest2

从 1.7 版本开始不推荐使用

Python 2.7对`unittest`库引入了一些重大更改，添加了一些非常有用的功能。为了确保每个Django项目都能从这些新功能中受益，Django使用了一个Python 2.7的`unittest`副本，用于Python 2.6兼容性。

由于Django不再支持2.7以前版本的Python版本，因此不推荐使用`django.utils.unittest`。只需使用`unittest`即可。

下面是一个对`django.test.TestCase`子类化的实例，前者同时也是`unittest`的子类。If your tests rely on database access such as creating or querying models, be sure to create your test classes as subclasses of django.test.TestCase rather than unittest.TestCase.Using unittest.TestCase avoids the cost of running each test in a transaction and flushing the database, but if your tests interact with the database their behavior will vary based on the order that the test runner executes them. This can lead to unit tests that pass when run in isolation but fail when run in a suite

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    def setUp(self):
        Animal.objects.create(name="lion", sound="roar")
        Animal.objects.create(name="cat", sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        lion = Animal.objects.get(name="lion")
        cat = Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(), 'The lion says "roar"')
        self.assertEqual(cat.speak(), 'The cat says "meow"')
```

当你 *run your tests*,测试工具的默认操作是寻找所有的测试用例（即 `unittest.` 的子类TestCase）在名称以 `test` 开头的任何文件中，自动从这些测试用例构建测试套件，并运行该套件。

更多关于 `unittest` 的细节，请参阅 Python文档。

警告

如果您的测试依赖于数据库访问，例如创建或查询模型，请务必将测试类创建为 `django.test.TestCase` 的子类，而不是 `unittest.` 测试用例。

使用 `unittest.` TestCase避免了在事务中运行每个测试和刷新数据库的成本，但是如果您的测试与数据库交互，它们的行为将根据测试运行器执行它们的顺序而变化。这可能导致单元测试在孤立运行时传递，但在套件中运行时失败。

运行测试

完成测试后，请使用您的项目 `manage.py` 实用程序的 `test` 命令运行它们：

```
$ ./manage.py test
```

测试发现基于unittest模块的*built-in test discovery*。默认情况下，这将在当前工作目录下名为“test * .py”的任何文件中发现测试。

您可以通过向 `./ manage.py 测试` 提供任意数量的“测试标签”来指定要运行的特定测试。每个测试标签可以是一个完整的Python虚线路径到一个包，模块， `TestCase` 子类或测试方法。例如：

```
# Run all the tests in the animals.tests module
$ ./manage.py test animals.tests

# Run all the tests found within the 'animals' package
$ ./manage.py test animals

# Run just one test case
$ ./manage.py test animals.tests.AnimalTestCase

# Run just one test method
$ ./manage.py test animals.tests.AnimalTestCase.test_animals_can
_speak
```

您还可以提供目录的路径，以发现该目录下面的测试：

```
$ ./manage.py test animals/
```

You can specify a custom filename pattern match using the `-p` (or `--pattern`) option, if your test files are named differently from the `test*.py` pattern:

```
$ ./manage.py test --pattern="tests_*.py"
```

如果正在测试的时候按下了 `Ctrl-C` ，测试程序会等待当前正在运行的测试完毕之后再退出。在正常退出期间，测试运行程序将输出任何测试失败的详细信息，报告运行了多少测试以及遇到了多少错误和失败，并像往常一样销毁任何测试数据库。因此，如果您忘记传递 `--failfast` 选项，注意某些测试意外失败，并想要获取有关失败的详细信息，那么按 `Ctrl-C` 而无需等待完整的测试运行完成。

如果您不想等待当前运行的测试完成，您可以再次按 `Ctrl-C` ，测试运行将立即停止，但不会正常停止。不会报告在中断之前运行的测试的详细信息，并且将不会销毁由运行创建的任何测试数据库。

测试时启用警告

建议您在启用Python警告后执行测试： `python -Wall manage.py 测试 t4 >` 。 `-Wall` 标志告诉Python显示弃用警告。与许多其他Python库一样，Django使用这些警告来标记特性何时消失。它也可以标记你的代码中的区域，不是严格错误，但可以从更好的实现中受益。

测试数据库

需要数据库的测试（即模型测试）不会使用您的“真实”（生产）数据库。为测试创建单独的空白数据库。

无论测试通过还是失败，当所有测试都已执行时，测试数据库将被销毁。

New in Django 1.8:

您可以通过向测试命令添加 `--keepdb` 标志来防止测试数据库被破坏。这将在运行之间保留测试数据库。如果数据库不存在，它将首先被创建。任何迁移也将应用，以保持最新。

默认情况下，测试数据库通过在 `DATABASES` 中定义的数据库的 `NAME` 设置的前面加上 `test_` 当使用SQLite数据库引擎时，测试将默认使用内存数据库（即，数据库将在内存中创建，完全绕过文件系统）。如果要使用其他数据库名称，请在 `DATABASES` 中的任何给定数据库的 `TEST` 字典中指定 `NAME` 。

Changed in Django 1.7:

在PostgreSQL上， `USER` 还需要对内置的 `postgres` 数据库进行读取访问。

除了使用单独的数据库，否则测试运行程序将使用您的设置文件中的所有相同的数据库设置： `ENGINE` ， `USER` ， `HOST` 测试数据库由 `USER` 指定的用户创建，因此您需要确保给定的用户帐户具有在系统上创建新数据库的足够权限。

要对测试数据库的字符编码进行细粒度控制，请使用 `CHARSET` TEST选项。如果您使用MySQL，还可以使用 `COLLATION` 选项来控制测试数据库使用的特定归类。有关这些和其他高级设置的详细信息，请参阅*settings documentation*。

如果使用带有Python 3.4+和SQLite 3.7.13+的SQLite内存数据库，将启用共享缓存，因此您可以编写测试，以便在线程之间共享数据库。

Changed in Django 1.7:

在 `TEST` 数据库设置中的不同选项在数据库设置字典中是单独的选项，前缀为 `TEST_` 。

New in Django 1.8:

添加了使用如上所述的共享缓存的SQLite的能力。

在运行测试时从生产数据库中查找数据？

如果您的代码在编译模块时尝试访问数据库，则会在测试数据库设置之前发生，可能会出现意外结果。例如，如果您在模块级代码中有数据库查询，并且存在真实数据库，生产数据可能会污染您的测试。这是一个坏主意，在你的代码中有这样的导入时数据库查询 - 重写你的代码，以便它不这样做。

New in Django 1.7:

这也适用于 `ready()` 的自定义实现。

也可以看看

advanced multi-db testing topics。

执行测试的顺序

为了保证所有 `TestCase` 代码以干净的数据库开始，Django测试运行器以下列方式重新排序测试：

- 所有 `TestCase` 子类首先运行。
- 然后，运行所有其他基于Django的测试（基于 `SimpleTestCase` 的测试用例，包括 `TransactionTestCase` ），而不保证或强制执行其中的特定顺序。
- 然后是任何其他 `unittest.` 运行可能更改数据库而不将其恢复到其原始状态的TestCase测试（包括doctests）。

注意

测试的新排序可能揭示对测试用例排序的意外依赖。这是依赖于通过给定的 `TransactionTestCase` 测试在数据库中保留的状态的doctests的情况，它们必须被更新以能够独立运行。

New in Django 1.8:

通过将 `--reverse` 传递到测试命令，可以反转组内的执行顺序。这可以帮助确保您的测试彼此独立。

回滚仿真

迁移中加载的任何初始数据只能在 `TestCase` 测试中使用，而不能在 `TransactionTestCase` 测试中使用，并且只能在支持事务的后端（最重要的例外是MyISAM）。对于依赖于 `TransactionTestCase` （例如 `LiveServerTestCase` 和 `StaticLiveServerTestCase` ）的测试也是如此。

Django can reload that data for you on a per-testcase basis by setting the `serialized_rollback` option to `True` in the body of the `TestCase` or `TransactionTestCase` , but note that this will slow down that test suite by approximately 3x.

第三方应用程序或针对MyISAM开发的应用程序需要进行设置；但是，一般来说，您应该针对事务数据库开发自己的项目，并且对大多数测试使用 `TestCase` ，因此不需要此设置。

初始序列化通常非常快速，但如果您希望从此过程中排除某些应用（并加快测试运行速度），您可以将这些应用添加到 `TEST_NON_SERIALIZED_APPS` 。

没有迁移的应用程序不受影响； `initial_data` fixtures像往常一样重新载入。

其他测试条件

无论配置文件中 `DEBUG` 设置的值如何，所有Django测试都将运行 `DEBUG` = False。这是为了确保您的代码的观察输出与生产设置中将看到的匹配。

缓存不会在每次测试后清除，如果您在生产环境中运行测试，运行“manage.py test fooapp”可以将测试中的数据插入到实时系统的缓存中，因为与数据库不同，单独的“测试缓存”不是用过的。此行为可能会在未来更改。

了解测试输出

当您运行测试时，您会看到一些消息，因为测试运行器准备自己。您可以使用命令行上的 `verbosity` 选项控制这些消息的详细程度：

```
Creating test database...
Creating table myapp_animal
Creating table myapp_mineral
Loading 'initial_data' fixtures...
No fixtures found.
```

这告诉你测试运行器正在创建一个测试数据库，如上一节所述。

一旦创建了测试数据库，Django将运行你的测试。如果一切顺利，你会看到这样的：

```
----------------------------------------------------------------
------
Ran 22 tests in 0.221s

OK
```

但是，如果有测试失败，您将看到有关哪些测试失败的完整详细信息：

```
================================================================
======
FAIL: test_was_published_recently_with_future_poll (polls.tests.
PollMethodTests)
----------------------------------------------------------------
------
Traceback (most recent call last):
  File "/dev/mysite/polls/tests.py", line 16, in test_was_publis
hed_recently_with_future_poll
    self.assertEqual(future_poll.was_published_recently(), False
)
AssertionError: True != False

----------------------------------------------------------------
------
Ran 1 test in 0.003s

FAILED (failures=1)
```

此错误输出的完整解释超出了本文档的范围，但它是非常直观。有关详细信息，请参阅Python的 `unittest` 库文档。

请注意，对于任何数量的失败和错误测试，测试运行程序脚本的返回码为1。如果所有测试通过，返回码为0。如果您在shell脚本中使用测试运行程序脚本并需要在该级别上测试成功或失败，则此功能非常有用。

加快测试速度

在最近的Django版本中，默认密码hasher设计相当慢。如果在测试期间您正在认证许多用户，则可能需要使用自定义设置文件，并将 `PASSWORD_HASHERS` 设置为更快的散列算法：

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.MD5PasswordHasher',
)
```

不要忘记也可以在 `PASSWORD_HASHERS` 中包含灯具中使用的任何散列算法（如果有的话）。

测试工具

Django提供了一组小工具，在写测试时派上用场。

测试客户端

测试客户端是一个Python类，作为一个虚拟的Web浏览器，允许您测试您的视图，并与您的Django供电的应用程序以编程方式交互。

你可以用测试客户端做的一些事情是：

- 模拟对URL的GET和POST请求，并观察响应 - 从低级HTTP（结果头和状态代码）到页面内容的一切。
- 查看重定向链（如果有），并在每个步骤中检查网址和状态代码。
- 测试给定的请求是否由给定的Django模板呈现，其中模板上下文包含某些值。

请注意，测试客户端不是要替代Selenium或其他“浏览器内”框架。Django的测试客户端有不同的焦点。简而言之：

- 使用Django的测试客户端来确定正在渲染正确的模板，并且传递正确的上下文数据。
- 使用Selenium等浏览器框架测试网页的呈现的 HTML和行为，即JavaScript功能。Django还为这些框架提供特殊支持；有关详细信息，请参阅 `LiveServerTestCase` 一节 。

一个全面的测试套件应该使用两种测试类型的组合。

概述和一个快速示例

要使用测试客户端，请实例化 `django.test.Client` 并检索网页：

```
>>> from django.test import Client
>>> c = Client()
>>> response = c.post('/login/', {'username': 'john', 'password'
: 'smith'})
>>> response.status_code
200
>>> response = c.get('/customer/details/')
>>> response.content
'<!DOCTYPE html...'
```

正如此示例所建议的，您可以在Python交互式解释器的会话中实例化 `Client` 。

请注意测试客户端如何工作的几个重要的事情：

- 测试客户端不要求Web服务器正在运行。事实上，它将运行很好，没有Web服务器运行在所有！这是因为它避免了HTTP的开销，直接处理Django框架。这有助于使单元测试快速运行。
- 检索网页时，请记住指定网址的路径，而不是整个网域。例如，这是正确的：

  ```
  &gt;&gt;&gt; c.get('/login/')
  ```

  这是不正确的：

  ```
  &gt;&gt;&gt; c.get('http://www.example.com/login/')
  ```

  测试客户端无法检索不受您的Django项目驱动的网页。如果需要检索其他网页，请使用Python标准库模块，例如 `urllib` 。
- 要解析网址，测试客户端将使用您的 `ROOT_URLCONF` 设置指向的任何URLconf。
- 虽然上面的例子可以在Python交互式解释器中工作，但是测试客户端的一些功能，特别是与模板相关的功能，只有在测试运行时才可用。

  这样做的原因是，Django的测试运行器执行一些黑魔法，以确定哪个模板由给定的视图加载。这个黑魔法（本质上是Django的模板系统在内存中的修补）只发生在测试运行期间。
- 默认情况下，测试客户端将禁用由您的站点执行的任何CSRF检查。

  如果由于某种原因，您希望测试客户端执行CSRF检查，您可以创建实施CSRF检查的测试客户端的实例。为此，在构建客户端时传递 `enforce_csrf_checks` 参数：

  ```
  &gt;&gt;&gt; from django.test import Client
  &gt;&gt;&gt; csrf_client = Client(enforce_csrf_checks=True)
  ```

提出请求

使用 `django.test.Client` 类发出请求。

class `Client` (*enforce_csrf_checks=False*, ***defaults*)

它在建设时不需要论证。但是，您可以使用关键字参数指定一些默认标头。例如，这会在每个请求中发送 `User-Agent` HTTP标头：

```
&gt;&gt;&gt; c = Client(HTTP_USER_AGENT='Mozilla/5.0')
```

传递给 `get()` ， `post()` 等的 `extra` 关键字参数的值优先于传递给类构造函数的默认值。

`enforce_csrf_checks` 参数可用于测试CSRF保护（参见上文）。

一旦您有一个 `Client` 实例，您就可以调用以下任何方法：

`get` (*path*, *data=None*, *follow=False*, *secure=False*, ***extra*)

New in Django 1.7:

已添加 `secure` 参数。

在提供的 `path` 上发出GET请求，并返回 `Response` 对象，下面将对此进行说明。

`data` 字典中的键值对用于创建GET数据有效内容。例如：

```
&gt;&gt;&gt; c = Client()
&gt;&gt;&gt; c.get('/customers/details/', {'name': 'fred', 'age'
: 7})
```

...将导致GET请求的求值等效于：

```
/customers/details/?name=fred&age=7
```

`extra` 关键字arguments参数可用于指定要在请求中发送的标头。例如：

```
&gt;&gt;&gt; c = Client()
&gt;&gt;&gt; c.get('/customers/details/', {'name': 'fred', 'age'
: 7},
...       HTTP_X_REQUESTED_WITH='XMLHttpRequest')
```

...将HTTP标头 `HTTP_X_REQUESTED_WITH` 发送到详细信息视图，这是一个测试使用 `django.http.HttpRequest.is_ajax()` 方法的代码路径的好方法。

CGI规范

通过 `**extra` 发送的标头应遵循CGI规范。例如，模拟从浏览器到服务器的HTTP请求中发送的不同“主机”标头应作为 `HTTP_HOST` 传递。

如果您已经有以URL编码形式的GET参数，则可以使用该编码，而不是使用data参数。例如，先前的GET请求也可以被提出为：

```
&gt;&gt;&gt; c = Client()
&gt;&gt;&gt; c.get('/customers/details/?name=fred&age=7')
```

如果您提供的网址包含编码的GET数据和数据参数，则数据参数优先。

如果您将 `follow` 设置为 `True` ，则客户端将跟踪任何重定向，并且将在包含中间网址元组的响应对象中设置 `redirect_chain` 属性和状态代码。

如果您有重定向到 `/next/` 的网址 `/redirect_me/` ，则重定向到 `/final/` ，这是您会看到的：

```
&gt;&gt;&gt; response = c.get('/redirect_me/', follow=True)
&gt;&gt;&gt; response.redirect_chain
[('http://testserver/next/', 302), ('http://testserver/final/',
302)]
```

如果您将 `secure` 设置为 `True` ，则客户端将模拟HTTPS请求。

`post` (*path*, *data=None*, *content_type=MULTIPART_CONTENT*, *follow=False*, *secure=False*, ***extra*)

在提供的 `path` 上发出POST请求，并返回 `Response` 对象，这在下面进行了说明。

`data` 字典中的键值对用于提交POST数据。例如：

```
&gt;&gt;&gt; c = Client()
&gt;&gt;&gt; c.post('/login/', {'name': 'fred', 'passwd': 'secre
t'})
```

...将导致对此URL的POST请求的评估：

```
/login/
```

...用这个POST数据：

```
name=fred&passwd=secret
```

如果您为XML有效内容提供 `content_type` （例如*text / xml*），则 `data` 的内容将作为POST请求，使用HTTP `Content-Type` 标头中的 `content_type` 。

如果不为 `content_type` 提供值，则 `data` 中的值将以内容类型*multipart / form-data*传输。在这种情况下， `data` 中的键值对将被编码为一个多部分消息，并用于创建POST数据有效负载。

要为给定键提交多个值 - 例如，为 `&lt； select multiple>` 指定选择，列表或元组。例如， `data` 的此值将为名为 `choices` 的字段提交三个选定值：

```
{'choices': ('a', 'b', 'd')}
```

提交文件是一种特殊情况。要发布文件，您只需要提供文件字段名作为键，以及要作为值上传的文件的文件句柄。例如：

```
&gt;&gt;&gt; c = Client()
&gt;&gt;&gt; with open('wishlist.doc') as fp:
...     c.post('/customers/wishes/', {'name': 'fred', 'attachmen
t': fp})
```

（名称 `attachment` 这里不相关；使用您的文件处理代码所期望的任何名称。）

您还可以提供任何类似文件的对象（例如， `StringIO` 或 `BytesIO` ）作为文件句柄。

New in Django 1.8:

添加了使用类文件对象的能力。

请注意，如果您希望对多个 `post()` 调用使用相同的文件句柄，则需要在文章之间手动重置文件指针。最简单的方法是在文件提供给 `post()` 之后手动关闭文件，如上所示。

您还应确保以允许读取数据的方式打开文件。如果您的文件包含二进制数据，如图像，这意味着您需要以 `rb` （读取二进制）模式打开该文件。

`extra` 参数的作用与 `Client.get()` 相同。

如果您通过POST请求的URL包含已编码的参数，则这些参数将在请求中可用。GET数据。例如，如果您提出请求：

```
&gt;&gt;&gt; c.post('/login/?visitor=true', {'name': 'fred', 'pa
sswd': 'secret'})
```

...处理此请求的视图可以询问请求。POST以检索用户名和密码，并且可以询问请求。GET以确定用户是否是访问者。

如果您将 `follow` 设置为 `True` ，则客户端将跟踪任何重定向，并且将在包含中间网址元组的响应对象中设置 `redirect_chain` 属性和状态代码。

如果您将 `secure` 设置为 `True` ，则客户端将模拟HTTPS请求。

`head` (*path*, *data=None*, *follow=False*, *secure=False*, ***extra*)

在提供的 `path` 上执行HEAD请求，并返回 `Response` 对象。此方法的工作方式与 `Client.get()` 一样，除了它之外，包括 `follow` ， `secure` 和 `extra`

`options` (*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)

在提供的 `path` 上执行OPTIONS请求，并返回 `Response` 对象。用于测试RESTful接口。

当提供 `data` 时，它用作请求主体，并且 `Content-Type` 头设置为 `content_type` 。

`follow` ， `secure` 和 `extra` 参数的作用与 `Client.get()` 相同。

`put` (*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)

在提供的 `path` 上发出PUT请求，并返回 `Response` 对象。用于测试RESTful接口。

当提供 `data` 时，它用作请求主体，并且 `Content-Type` 头设置为 `content_type` 。

`follow` ， `secure` 和 `extra` 参数的作用与 `Client.get()` 相同。

`patch` (*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)

在提供的 `path` 上发出PATCH请求，并返回 `Response` 对象。用于测试RESTful接口。

`follow` ， `secure` 和 `extra` 参数的作用与 `Client.get()` 相同。

`delete` (*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)

在提供的 `path` 上发出DELETE请求，并返回 `Response` 对象。用于测试RESTful接口。

当提供 `data` 时，它用作请求主体，并且 `Content-Type` 头设置为 `content_type` 。

`follow` ， `secure` 和 `extra` 参数的作用与 `Client.get()` 相同。

`trace` (*path*, *follow=False*, *secure=False*, ***extra*)

New in Django 1.8.

在提供的 `path` 上执行TRACE请求，并返回 `Response` 对象。用于模拟诊断探头。

与其他请求方法不同，为了符合 **RFC 2616**， `data` 不作为关键字参数提供，这要求TRACE请求不应有一个实体体。

`follow` ， `secure` 和 `extra` 参数的作用与 `Client.get()` 相同。

`login` (***credentials*)

如果您的网站使用Django的*authentication system*，并且您处理用户登录，则可以使用测试客户端的 `login()` 方法来模拟用户登录网站的效果。

调用此方法后，测试客户端将拥有通过任何可能构成视图一部分的基于登录的测试所需的所有Cookie和会话数据。

`credentials` 参数的格式取决于您使用的*authentication backend*（由您的 `AUTHENTICATION_BACKENDS` 设置配置）。如果您使用的是由Django提供的标准认证后端（ `ModelBackend` ），则 `credentials` 应该是用户的用户名和密码，

```
&gt;&gt;&gt; c = Client()
&gt;&gt;&gt; c.login(username='fred', password='secret')

# Now you can access a view that's only available to logged-in u
sers.
```

如果您使用的是其他身份验证后端，则此方法可能需要不同的凭据。它需要您的后端的 `authenticate()` 方法需要的凭据。

`login()` returns `True` if it the credentials were accepted and login was successful.

最后，您需要记住创建用户帐户，然后才能使用此方法。如上所述，测试运行器是使用测试数据库执行的，默认情况下不包含用户。因此，在生产站点上有效的用户帐户将无法在测试条件下工作。您需要创建用户作为测试套件的一部分 - 手动（使用Django模型API）或测试夹具。请记住，如果您希望测试用户拥有密码，则不能通过直接设置password属性来设置用户的密码 - 您必须使用 `set_password()` 函数来存储正确的散列密码。或者，您可以使用 `create_user()` 助手方法创建具有正确散列密码的新用户。

`logout` ()

如果您的网站使用Django的*authentication system*，则可以使用 `logout()` 方法模拟用户从您的网站注销的效果。

调用此方法后，测试客户端将所有Cookie和会话数据清除为默认值。后续请求将显示为来自 `AnonymousUser` 。

测试响应

`get()` 和 `post()` 方法都会返回 `Response` 对象。此 `Response` 对象不是与Django视图返回的 `HttpResponse` 对象相同；测试响应对象具有一些对于测试代码验证有用的附加数据。

具体来说， `Response` 对象具有以下属性：

class `Response`

`client`

用于生成导致响应的请求的测试客户端。

`content`

响应的主体，作为字符串。这是视图呈现的最终页面内容，或任何错误消息。

`context`

用于呈现产生响应内容的模板的模板 `Context` 实例。

如果呈现的页面使用多个模板，则 `context` 将是 `Context` 对象的列表，按照它们的呈现顺序。

无论渲染期间使用的模板数量如何，都可以使用 `[]` 运算符检索上下文值。例如，可以使用以下方式检索上下文变量 `name` ：

```
&gt;&gt;&gt; response = client.get('/foo/')
&gt;&gt;&gt; response.context['name']
'Arthur'
```

`request`

刺激响应的请求数据。

`wsgi_request`

New in Django 1.7.

由生成响应的测试处理程序生成的 `WSGIRequest` 实例。

`status_code`

响应的HTTP状态，作为整数。有关HTTP状态代码的完整列表，请参见 **RFC 2616**。

`templates`

用于渲染最终内容的 `Template` 实例列表，按渲染顺序排列。对于列表中的每个模板，如果从文件加载模板，请使用 `template.name` 获取模板的文件名。（名称是一个字符串，例如 `'admin/index.html'` 。）

`resolver_match`

New in Django 1.8:

响应的实例 `ResolverMatch` 。例如，您可以使用 `func` 属性验证提供响应的视图：

```
# my_view here is a function based view
self.assertEqual(response.resolver_match.func, my_view)

# class based views need to be compared by name, as the function
s
# generated by as_view() won't be equal
self.assertEqual(response.resolver_match.func.__name__, MyView.a
s_view().__name__)
```

如果找不到给定的URL，访问此属性将引发 `Resolver404` 异常。

您还可以在响应对象上使用字典语法查询HTTP标头中的任何设置的值。例如，您可以使用 `response['Content-Type']` 确定响应的内容类型。

例外

如果将测试客户端指向引发异常的视图，那么该异常将在测试用例中可见。然后，您可以使用标准 `尝试 ...` 除了块或 `assertRaises()` 来测试异常。

对测试客户端不可见的唯一例外是 `Http404` ， `PermissionDenied` ， `SystemExit` 和 `SuspiciousOperation` 。Django在内部捕获这些异常并将它们转换为适当的HTTP响应代码。在这些情况下，您可以在测试中检查 `response.status_code` 。

持久状态

测试客户端是有状态的。如果响应返回cookie，那么该cookie将存储在测试客户端中，并与所有后续的 `get()` 和 `post()` 请求一起发送。

不遵循这些cookie的过期政策。如果您希望Cookie过期，请手动删除或创建新的 `Client` 实例（这将有效删除所有Cookie）。

测试客户机具有存储持久状态信息的两个属性。您可以作为测试条件的一部分访问这些属性。

`Client.``cookies`

Python `SimpleCookie` 对象，包含所有客户端Cookie的当前值。有关更多信息，请参阅 `http.cookies` 模块的文档。

`Client.``session`

包含会话信息的类字典对象。有关详细信息，请参阅*session documentation*。

要修改会话然后保存它，它必须首先存储在变量中（因为每次访问此属性时都会创建一个新的 `SessionStore` ）：

```
def test_something(self):
    session = self.client.session
    session['somekey'] = 'test'
    session.save()
```

例

以下是使用测试客户端的简单单元测试：

```
import unittest
from django.test import Client

class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs a client.
        self.client = Client()

    def test_details(self):
        # Issue a GET request.
        response = self.client.get('/customer/details/')

        # Check that the response is 200 OK.
        self.assertEqual(response.status_code, 200)

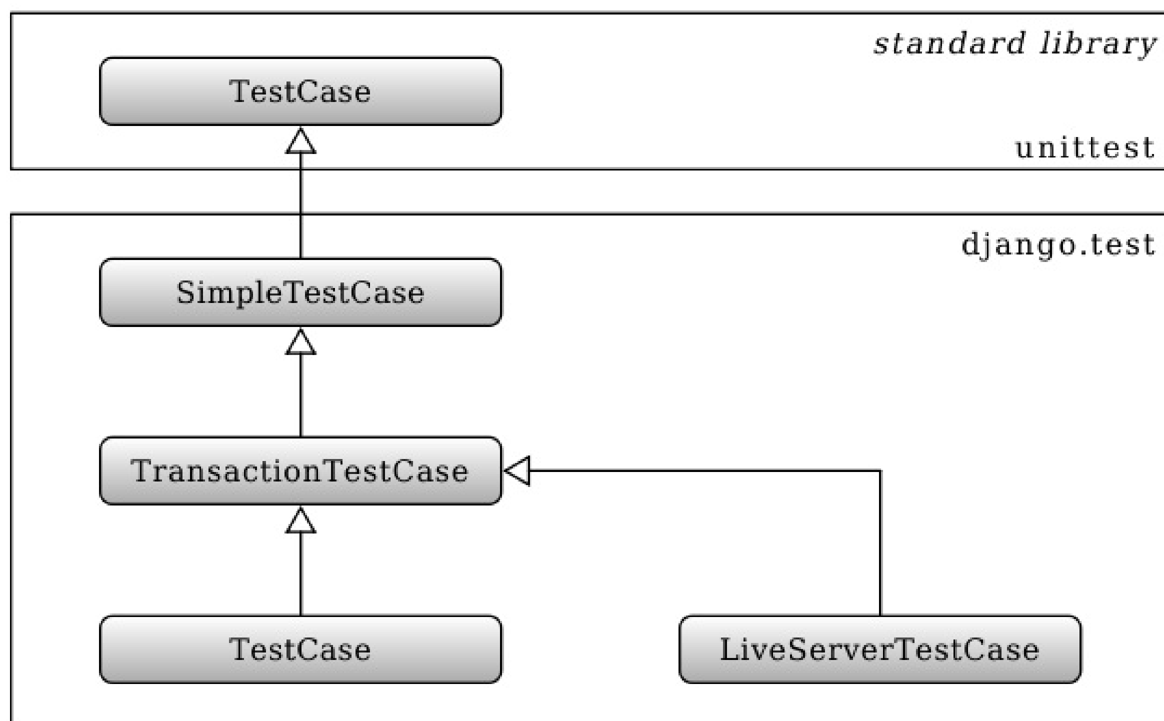
        # Check that the rendered context contains 5 customers.
        self.assertEqual(len(response.context['customers']), 5)
```

也可以看看

`django.test.RequestFactory`

提供测试用例类

正常的Python单元测试类扩展了 `unittest.` 测试用例。Django提供了这个基类的一些扩展：



Django单元测试类的层次结构

SimpleTestCase

class `SimpleTestCase`

`unittest.` TestCase，它扩展它与一些基本功能，如：

- 保存和恢复Python警告机制状态。
- Some useful assertions like:
  - Checking that a callable .
  - 测试表单字段 `rendering and error treatment` 。
  - 测试 `HTML responses for the presence/lack of a given fragment` 。
  - 验证模板 `has/hasn't been used to generate a given response content` 。
  - 验证HTTP `redirect` 是由应用执行的。
  - 稳健地测试两个 `HTML fragments` 的等式/不等式或 `containment` 。
  - 稳健地测试两个 `XML fragments` 的相等/不等。
  - 稳健地测试两个 `JSON fragments` 的相等性。
- 使用*modified settings*运行测试的能力。
- 使用 `client` `Client` 。
- 自定义测试时间 `URL maps` 。

如果您需要任何其他更复杂和重量级的Django特定功能，如：

- 测试或使用ORM。
- 数据库 `fixtures` 。
- 根据数据库后端功能测试*skipping based on database backend features*。
- 其余的专用 `assert*` 方法。

那么您应该改用 `TransactionTestCase` 或 `TestCase` 。

`SimpleTestCase` 继承自 `unittest.` 测试用例。

警告

`SimpleTestCase` 及其子类（例如 `TestCase` ，...）依赖于 `setUpClass()` 和 `tearDownClass()` 执行一些类的初始化（例如覆盖设置）。如果需要重写这些方法，不要忘记调用 `super` 实现：

```
class MyTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(MyTestCase, cls).setUpClass()     # Call parent fi
rst
        ...

    @classmethod
    def tearDownClass(cls):
        ...
        super(MyTestCase, cls).tearDownClass()  # Call parent la
st
```

TransactionTestCase

class `TransactionTestCase`

Django的 `TestCase` 类（如下所述）利用数据库事务工具来加快在每个测试开始时将数据库重置为已知状态的过程。然而，这样做的一个后果是，一些数据库行为不能在Django `TestCase` 类中测试。例如，您不能测试一个代码块是否在事务中执行，如使用 `select_for_update()` 时所需。在这些情况下，您应该使用 `TransactionTestCase` 。

Changed in Django 1.8:

在旧版本的Django中，事务提交和回滚的影响无法在 `TestCase` 中测试。随着Django 1.8中旧式事务管理的弃用循环的完成，在 `TestCase` 中不再禁用事务管理命令（例如 `transaction.commit()` ）。

`TransactionTestCase` 和 `TestCase` 是完全相同的，除了数据库重置为已知状态的方式以及测试代码测试提交和回滚效果的能力：

- 通过截断所有表，测试运行后， `TransactionTestCase` 会重置数据

库。 `TransactionTestCase` 可以调用提交和回滚，并观察这些调用对数据库的影响。
- 另一方面， `TestCase` 在测试后不会截断表。相反，它将测试代码包含在测试结束时回滚的数据库事务中。这保证在测试结束时的回滚将数据库恢复到其初始状态。

警告

在不支持回滚的数据库上运行的 `TestCase` （例如，具有MyISAM存储引擎的MySQL）以及 `TransactionTestCase` 的所有实例将在测试结束时回滚从测试数据库中删除所有数据，并重新加载应用程序的初始数据而不进行迁移。

迁移*will not see their data reloaded*；如果您需要此功能（例如，第三方应用应启用此功能），您可以设置 `serialized_rollback = True t0 >在` TestCase `内。`

`TransactionTestCase` 继承自 `SimpleTestCase` 。

测试用例

class `TestCase`

这个类提供了一些额外的功能，可以用来测试网站

正常转换 `unittest.` TestCase到Django `TestCase` 很简单：只需从 `'unittest.` TestCase'to `'django.test.TestCase'` 。所有标准的Python单元测试功能将继续可用，但将增加一些有用的补充，包括：

- 自动加载夹具。
- 将测试包含在两个嵌套的 `atomic` 块中：一个用于整个类，一个用于每个测试。
- 创建一个TestClient实例。
- Django特定的断言用于测试重定向和形式错误。

classmethod `TestCase.``setUpTestData` ()

New in Django 1.8.

上述类级别 `atomic` 块允许在类级别创建初始数据，对于整个 `TestCase` 一次。与使用 `setUp()` 相比，此技术允许更快的测试。

例如：

```
from django.test import TestCase

class MyTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up data for the whole TestCase
        cls.foo = Foo.objects.create(bar="Test")
        ...

    def test1(self):
        # Some test using self.foo
        ...

    def test2(self):
        # Some other test using self.foo
        ...
```

注意，如果测试在没有事务支持的数据库上运行（例如，使用MyISAM引擎的MySQL），则在每次测试之前将调用 `setUpTestData()` ，否定速度优势。

警告

如果要测试一些特定的数据库事务行为，应该使用 `TransactionTestCase` 作为 `TestCase` 在 `atomic()` 块中测试执行。

`TestCase` 继承自 `TransactionTestCase` 。

LiveServerTestCase

class `LiveServerTestCase`

`LiveServerTestCase` 基本上与 `TransactionTestCase` 相同，具有一个额外的功能：它在设置的后台启动一个活动的Django服务器，并在拆卸时将其关闭。这允许使用除了*Django dummy client*（例如，Selenium客户端）之外的自动测试客户端，在浏览器中执行一系列功能测试，并模拟真实用户的操作。

默认情况下，活动服务器的地址为 `'localhost:8081'` ，在测试期间可以使用 `self.live_server_url` 访问完整的URL。If you'd like to change the default address (in the case, for example, where the 8081 port is already taken) then you may pass a different one to the `test` command via the `--liveserver` option, for example:

```
$ ./manage.py test --liveserver=localhost:8082
```

更改默认服务器地址的另一种方法是在代码中某处设置 <cite>DJANGO*LIVE_TEST_SERVER_ADDRESS</cite>*环境变量（例如，在 *[_custom test runner*](advanced.html#topics-testing-test-runner)中）：

```
import os
os.environ['DJANGO_LIVE_TEST_SERVER_ADDRESS'] = 'localhost:8082'
```

在测试由多个进程并行运行的情况下（例如，在几个并发的连续集成构建的上下文中），进程将竞争同一个地址，因此您的测试可能随机失败并显示“地址已在使用中”错误。为了避免此问题，您可以传递逗号分隔的端口列表或端口范围（至少与潜在的并行进程数一样多）。例如：

```
$ ./manage.py test --liveserver=localhost:8082,8090-8100,9000-9200,7041
```

然后，在测试执行期间，每个新的活测试服务器将尝试每个指定的端口，直到找到一个可用的并且接受它。

为了演示如何使用,让我们编写一个简单的Selenium测试。 首先呢,你需要用pip命令安装 selenium package 到你的Python路径里面:

```
$ pip install selenium
```

然后，向应用程序的测试模块添加 `LiveServerTestCase` 测试（例如： `myapp/tests.py` ）。此测试的代码可能如下所示：

```
from django.test import LiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver

class MySeleniumTests(LiveServerTestCase):
    fixtures = ['user-data.json']

    @classmethod
    def setUpClass(cls):
        super(MySeleniumTests, cls).setUpClass()
        cls.selenium = WebDriver()

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super(MySeleniumTests, cls).tearDownClass()

    def test_login(self):
        self.selenium.get('%s%s' % (self.live_server_url, '/logi
n/'))
        username_input = self.selenium.find_element_by_name("use
rname")
        username_input.send_keys('myuser')
        password_input = self.selenium.find_element_by_name("pas
sword")
        password_input.send_keys('secret')
        self.selenium.find_element_by_xpath('//input[@value="Log
 in"]').click()
```

最后，您可以运行测试如下：

```
$ ./manage.py test myapp.tests.MySeleniumTests.test_login
```

此示例将自动打开Firefox，然后转到登录页面，输入凭据并按"登录"按钮。Selenium提供其他驱动程序，以防您没有安装Firefox或希望使用其他浏览器。上面的例子只是Selenium客户端可以做的一小部分；有关详细信息，请参阅完整参考。

Changed in Django 1.7:

在旧版本中， `LiveServerTestCase` 依赖*staticfiles contrib app*在测试执行期间透明地提供静态文件。此功能已移至 `StaticLiveServerTestCase` 子类，因此如果您需要*the original behavior*，请使用该子类。

`LiveServerTestCase` 现在只需在 `STATIC_ROOT` 下的 `STATIC_URL` 发布文件系统的内容。

注意

当使用内存中的SQLite数据库来运行测试时，同一数据库连接将由两个并行线程共享：运行活动服务器的线程和运行测试用例的线程。重要的是防止两个线程通过这个共享连接同时进行数据库查询，因为这可能会随机导致测试失败。所以你需要确保这两个线程不会同时访问数据库。特别是，这意味着在某些情况下（例如，在单击链接或提交表单之后），您可能需要检查Selenium是否收到响应，并且在继续执行进一步的测试之前加载下一页。例如，通过使Selenium等待，直到在响应中找到 `<body>` HTML标记（需要Selenium> 2.13）：

```
def test_login(self):
    from selenium.webdriver.support.wait import WebDriverWait
    timeout = 2
    ...
    self.selenium.find_element_by_xpath('//input[@value="Log in"
]').click()
    # Wait until the response is received
    WebDriverWait(self.selenium, timeout).until(
        lambda driver: driver.find_element_by_tag_name('body'))
```

这里的棘手的事情是，真的没有一个“页面加载”，尤其是在现代Web应用程序中，在服务器生成初始文档后动态生成HTML。因此，简单地检查响应中是否存在 `<body>` 可能不一定适用于所有用例。有关详细信息，请参阅Selenium常见问题和Selenium文档。

测试用例特性

默认测试客户端

`SimpleTestCase.``client`

`django.test.*TestCase` 实例中的每个测试用例都可以访问Django测试客户端的实例。此客户端可以作为 `self.client` 访问。每个测试都重新创建此客户端，因此您不必担心从一个测试到另一个测试的状态（例如Cookie）。

这意味着，不是在每个测试中实例化 `Client` ：

```
import unittest
from django.test import Client

class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

...你可以参考 `self.client` ，像这样：

```
from django.test import TestCase

class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

自定义测试客户端

`SimpleTestCase.``client_class`

如果要使用不同的 `Client` 类（例如，具有自定义行为的子类），请使用 `client_class` 类属性：

```
from django.test import TestCase, Client

class MyTestClient(Client):
    # Specialized methods for your environment
    ...

class MyTest(TestCase):
    client_class = MyTestClient

    def test_my_stuff(self):
        # Here self.client is an instance of MyTestClient...
        call_some_test_code()
```

夹具装载

`TransactionTestCase.``fixtures`

如果数据库中没有任何数据，则对于数据库支持的Web站点的测试用例没有多大用处。为了方便将测试数据放入数据库，Django的自定义 `TransactionTestCase` 类提供了加载**fixtures**的方法。

fixture是Django知道如何导入到数据库中的数据集合。例如，如果您的网站有用户帐户，您可能会设置一个假的用户帐户，以便在测试期间填充您的数据库。

创建fixture的最直接的方法是使用 `manage.py dumpdata` 命令。这假定您的数据库中已经有一些数据。有关详细信息，请参阅 `dumpdata documentation` 。

注意

如果您曾经运行过 `manage.py migrate` ，您已经使用了一个甚至不知道它的灯具！当您第一次在数据库中调用 `migrate` 时，Django会安装一个名为 `initial_data` 的夹具。这为您提供了一种使用任何初始数据填充新数据库的方法，例如默认的一组类别。

可以使用 `manage.py loaddata` 命令手动安装具有其他名称的装置。

初始SQL数据和测试

Django提供了将初始数据插入模型的第二种方法 - *custom SQL hook*。然而，该技术_不能用于提供用于测试目的的初始数据。_Django的测试框架在每次测试后刷新测试数据库的内容；因此，使用自定义SQL钩子添加的任何数据都将丢失。

Once you've created a fixture and placed it in a `fixtures` directory in one of your `INSTALLED_APPS` , you can use it in your unit tests by specifying a `fixtures` class attribute on your `django.test.TestCase` subclass:

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
        # Test definitions as before.
        call_setup_methods()

    def testFluffyAnimals(self):
        # A test that uses the fixtures.
        call_some_test_code()
```

这里具体是什么会发生：

- 在每个测试用例开始时，在运行 `setUp()` 之前，Django将刷新数据库，将数

据库返回到调用 `migrate` 之后的状态。
- 然后，安装所有命名的夹具。在这个例子中，Django将安装任何名为 `mammals` 的JSON夹具，然后安装任何名为 `birds` 的夹具。有关定义和安装灯具的更多详细信息，请参阅 `loaddata` 文档。

对测试用例中的每个测试重复这个刷新/装载过程，因此您可以确定测试的结果不会受到另一个测试或测试执行的顺序的影响。

默认情况下，fixture仅加载到 `default` 数据库中。如果您使用多个数据库并设置 `multi_db=True` ，fixture将被加载到所有数据库。

URLconf配置

`SimpleTestCase.``urls`

自1.8版起已弃用：请改用URLconf配置 `@override_settings(ROOT_URLCONF=...)` 。

如果您的应用程序提供了视图，您可能需要包括使用测试客户端来执行这些视图的测试。但是，最终用户可以在自己选择的任何URL上自由地在应用程序中部署视图。

为了为测试提供可靠的URL空间， `django.test.*TestCase` 类提供了在测试套件执行期间自定义URLconf配置的功能。如果您的 `*TestCase` 实例定义了 `urls` 属性，则 `*TestCase` 将使用该属性的值作为 `ROOT_URLCONF` 。

例如：

```
from django.test import TestCase

class TestMyViews(TestCase):
    urls = 'myapp.test_urls'

    def testIndexPageView(self):
        # Here you'd test your view using ``Client``.
        call_some_test_code()
```

此测试用例将使用 `myapp.test_urls` 的内容作为测试用例持续时间的URLconf。

多数据库支持

`TransactionTestCase.``multi_db`

Django设置与设置文件中的 `DATABASES` 定义中定义的每个数据库对应的测试数据库。但是，运行Django TestCase所需的大部分时间由调用 `flush` 消耗，确保在每次测试运行开始时都有一个干净的数据库。如果您有多个数据库，则需要多次刷新（每个数据库一次），这可能是一个耗时的活动 - 特别是如果您的测试不需要测试多数据库活动。

作为优化，Django仅在每次测试运行开始时刷新 `default` 数据库。如果您的设置包含多个数据库，并且测试需要每个数据库都是干净的，则可以使用测试套件上的 `multi_db` 属性请求完全刷新。

例如：

```
class TestMyViews(TestCase):
    multi_db = True

    def testIndexPageView(self):
        call_some_test_code()
```

此测试用例将在运行 `testIndexPageView` 之前刷新所有测试数据库。

`multi_db` 标志还影响attr：<cite>TransactionTestCase.fixtures</cite>加载到哪些数据库中。默认情况下（当 `multi_db=False` 时），fixture仅加载到 `default` 数据库中。如果 `multi_db=True` ，fixture将加载到所有数据库中。

覆盖设置

警告

使用以下功能临时更改测试中的设置值。不要直接操作 `django.conf.settings` ，因为Django在这种操作后不会恢复原始值。

`SimpleTestCase.``settings` ()

为了测试目的，在运行测试代码之后临时更改设置并恢复为原始值通常很有用。对于这种用例，Django提供了一个标准的Python上下文管理器（参见 **PEP 343**），名为 `settings()`

```
from django.test import TestCase

class LoginTestCase(TestCase):

    def test_login(self):

        # First check for the default behavior
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/accounts/login/?next=/s
ekrit/')

        # Then override the LOGIN_URL setting
        with self.settings(LOGIN_URL='/other/login/'):
            response = self.client.get('/sekrit/')
            self.assertRedirects(response, '/other/login/?next=/
sekrit/')
```

此示例将覆盖 `with` 块中的代码的 `LOGIN_URL` 设置，然后将其值重置为之前的状态。

`SimpleTestCase.``modify_settings` ()

New in Django 1.7.

它可以证明难以重新定义包含值列表的设置。在实践中，添加或删除值通常就足够了。`modify_settings()` 上下文管理器使其变得容易：

```
from django.test import TestCase

class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        with self.modify_settings(MIDDLEWARE_CLASSES={
            'append': 'django.middleware.cache.FetchFromCacheMid
dleware',
            'prepend': 'django.middleware.cache.UpdateCacheMiddl
eware',
            'remove': [
                'django.contrib.sessions.middleware.SessionMiddl
eware',
                'django.contrib.auth.middleware.AuthenticationMi
ddleware',
                'django.contrib.messages.middleware.MessageMiddl
eware',
            ],
        }):
            response = self.client.get('/')
            # ...
```

对于每个操作，您可以提供值列表或字符串。当该值已经存在于列表中时， `append` 和 `prepend` 不起作用；当值不存在时， `remove` 。

`override_settings` ()

如果要覆盖测试方法的设置，Django提供 `override_settings()` 装饰器（请参阅 **PEP 318**）。它的使用像这样：

```
from django.test import TestCase, override_settings

class LoginTestCase(TestCase):

    @override_settings(LOGIN_URL='/other/login/')
    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekr
it/')
```

装饰器也可以应用于 `TestCase` 类：

```
from django.test import TestCase, override_settings

@override_settings(LOGIN_URL='/other/login/')
class LoginTestCase(TestCase):

    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekr
it/')
```

Changed in Django 1.7:

以前，`override_settings` 是从 `django.test.utils` 导入的。

`modify_settings` ()

New in Django 1.7.

同样，Django提供 `modify_settings()` 装饰器：

```
from django.test import TestCase, modify_settings

class MiddlewareTestCase(TestCase):

    @modify_settings(MIDDLEWARE_CLASSES={
        'append': 'django.middleware.cache.FetchFromCacheMiddlew
are',
        'prepend': 'django.middleware.cache.UpdateCacheMiddlewar
e',
    })
    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

装饰器也可以应用于测试用例类：

```
from django.test import TestCase, modify_settings

@modify_settings(MIDDLEWARE_CLASSES={
    'append': 'django.middleware.cache.FetchFromCacheMiddleware'
,
    'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
})
class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

注意

当给定一个类时，这些装饰器直接修改类并返回它；他们不创建并返回它的修改副本。因此，如果您尝试调整上述示例以将返回值分配给不同于 `LoginTestCase` 或 `MiddlewareTestCase` 的名称，您可能会惊讶地发现原始测试用例类仍然同样受到装饰师的影响。对于给定类，`modify_settings()` 始终应用于 `override_settings()` 之后。

警告

设置文件包含一些仅在Django内部初始化期间参考的设置。如果使用 `override_settings` 更改这些设置，如果通过 `django.conf.settings` 模块访问它，设置会更改，但是Django的内部访问方式不同。有效地，使用这些设置使用 `override_settings()` 或 `modify_settings()` 可能不会做你期望做的。

我们不建议更改 `DATABASES` 设置。改变 `CACHES` 设置是可能的，但如果你使用使用缓存的内部，如 `django.contrib.sessions` ，有点棘手。例如，您必须在使用缓存会话并覆盖 `CACHES` 的测试中重新初始化会话后端。

最后，避免将您的设置作为模块级常量别名，因为 `override_settings()` 将不适用于这些值，因为它们仅在首次导入模块时进行评估。

您还可以在设置被覆盖后通过删除设置来模拟缺少设置，如下所示：

```
@override_settings()
def test_something(self):
    del settings.LOGIN_URL
    ...
```

Changed in Django 1.7:

以前，您只能模拟删除显式覆盖的设置。

覆盖设置时，请确保处理您的应用代码使用缓存或类似功能的情况，即使设置更改也保留状态。Django提供了 `django.test.signals.setting_changed` 信号，允许您注册回调以清除设置，否则在更改设置时重置状态。

Django本身使用这个信号来重置各种数据：

Overridden settings	Data reset
USE_TZ, TIME_ZONE	Databases timezone
TEMPLATES	Template engines
SERIALIZATION_MODULES	Serializers cache
LOCALE_PATHS, LANGUAGE_CODE	Default translation and loaded translations
MEDIA_ROOT, DEFAULT_FILE_STORAGE	Default file storage

清空测试发件箱

如果您使用任何Django的自定义 `TestCase` 类，测试运行器将在每个测试用例开始时清除测试电子邮件发件箱的内容。

有关测试期间电子邮件服务的详细信息，请参阅下面的电子邮件服务。

断言

作为Python的正常 `unittest.` TestCase类实现诸如 `assertTrue()` 和 `assertEqual()` 之类的断言方法，Django的自定义 `TestCase` 类提供了许多有用的自定义断言方法用于测试Web应用程序：

大多数断言方法给出的失败消息可以使用 `msg_prefix` 参数定制。此字符串将作为断言生成的任何失败消息的前缀。这允许您提供其他详细信息，以帮助您确定测试套件中的故障位置和原因。

`SimpleTestCase.``assertRaisesMessage` (*expected_exception*, *expected_message*, *callable_obj=None*, **args*, ***kwargs*)

认为执行可调用 `callable_obj` 引发了 `expected_exception` 异常，并且此异常具有 `expected_message` 表示。任何其他结果报告为失败。类似于unittest的 `assertRaisesRegex()` ，区别在于 `expected_message` 不是正则表达式。

`SimpleTestCase.``assertFieldOutput` (*fieldclass*, *valid*, *invalid*, *field_args=None*, *field_kwargs=None*, *empty_value=''*)

断言表单字段在各种输入中正确运行。

Parameters:

- **fieldclass** - 要测试的字段的类。
- **valid** - 将有效输入映射到其预期清除值的字典。
- **invalid** - 将无效输入映射到一个或多个引发的错误消息的字典。
- **field_args** - 传递给实例化字段的arg。

- **field_kwargs** - 传递给实例化字段的kwargs。
- **empty_value** - `empty_values` 中输入的预期干净输出。

例如，以下代码测试 `EmailField` 接受 `a@a.com` 作为有效的电子邮件地址，但拒绝具有合理错误的 `aaa` 信息：

```
self.assertFieldOutput(EmailField, {'a@a.com': 'a@a.com'}, {'aaa': ['Enter a valid email address.']})
```

`SimpleTestCase.``assertFormError` (*response*, *form*, *field*, *errors*, *msg_prefix=''*)

断言窗体上的字段在表单上呈现时会提供所提供的错误列表。

`form` 是在模板上下文中给出的 `Form` 实例的名称。

`field` 是要检查的表单上的字段的名称。如果 `field` 的值为 `None` ，则将检查非字段错误（可通过 `form.non_field_errors()` 访问的错误）。

`errors` 是预期为表单验证结果的错误字符串或错误字符串列表。

`SimpleTestCase.``assertFormsetError` (*response*, *formset*, *form_index*, *field*, *errors*, *msg_prefix=''*)

断言 `formset` 在呈现时引发提供的错误列表。

`formset` 是在模板上下文中给出的 `Formset` 实例的名称。

`form_index` 是 `Formset` 内的表单编号。如果 `form_index` 的值为 `None` ，则将检查非格式错误（可通过 `formset.non_form_errors()` 访问的错误）。

`field` 是要检查的表单上的字段的名称。如果 `field` 的值为 `None` ，则将检查非字段错误（可通过 `form.non_field_errors()` 访问的错误）。

`errors` 是预期为表单验证结果的错误字符串或错误字符串列表。

`SimpleTestCase.``assertContains` (*response*, *text*, *count=None*, *status_code=200*, *msg_prefix=''*, *html=False*)

断言 `Response` 实例生成给定的 `status_code` ，并且该 `文本` 显示在响应的内容中。如果提供了 `count` ，则 `text` 必须在响应中出现正好 `count` 次数。

将 `html` 设置为 `True` ，将 `text` 作为HTML。与响应内容的比较将基于HTML语义，而不是逐个字符的等同。在大多数情况下忽略空白，属性排序不重要。有关详细信息，请参见 `assertHTMLEqual()` 。

`SimpleTestCase.``assertNotContains` (*response*, *text*, *status_code=200*, *msg_prefix=''*, *html=False*)

认为 `Response` 实例产生给定的 `status_code` ，并且 `text` 不会不会出现在响应的内容中。

将 `html` 设置为 `True` ，将 `text` 作为HTML。与响应内容的比较将基于HTML语义，而不是逐个字符的等同。在大多数情况下忽略空白，属性排序不重要。有关详细信息，请参见 `assertHTMLEqual()` 。

`SimpleTestCase.``assertTemplateUsed` (*response*, *template_name*, *msg_prefix=''*, *count=None*)

断言具有给定名称的模板用于呈现响应。

名称是一个字符串，例如 `'admin/index.html'` 。

New in Django 1.8:

count参数是一个整数，表示模板应该渲染的次数。默认值为 `None` ，表示模板应该呈现一次或多次。

你可以使用它作为上下文管理器，像这样：

```
with self.assertTemplateUsed('index.html'):
    render_to_string('index.html')
with self.assertTemplateUsed(template_name='index.html'):
    render_to_string('index.html')
```

`SimpleTestCase.``assertTemplateNotUsed` (*response*, *template_name*, *msg_prefix=''*)

Asserts that the template with the given name was *not* used in rendering the response.

您可以使用与 `assertTemplateUsed()` 相同的方式作为上下文管理器。

`SimpleTestCase.``assertRedirects` (*response*, *expected_url*, *status_code=302*, *target_status_code=200*, *host=None*, *msg_prefix=''*, *fetch_redirect_response=True*)

Asserts that the response returned a `status_code` redirect status, redirected to `expected_url` (including any `GET` data), and that the final page was received with `target_status_code` .

如果您的请求使用 `follow` 参数，则 `expected_url` 和 `target_status_code` 将是重定向链最后一点的网址和状态代码。

如果 `expected_url` 不包含一个（例如 `"/bar/"` ）， `host` 如果 `expected_url` 是包含主机的绝对网址（例如 `"http://testhost/bar/"` ），则 `host` 忽略。请注意，测试客户端不支持获取外部URL，但是如果您使用自定义HTTP主机进行测试（例如，使用 `Client(HTTP_HOST="testhost")`

New in Django 1.7.

如果 `fetch_redirect_response` 为 `False` ，则不会加载最后一页。由于测试客户端无法获取外部网址，因此如果 `expected_url` 不是您的Django应用程序的一部分，则此功能特别有用。

New in Django 1.7.

在两个URL之间进行比较时，会正确处理Scheme。如果在我们重定向到的位置中没有指定任何方案，则使用原始请求的方案。如果存在， `expected_url` 中的方案是用于进行比较的方案。

```SimpleTestCase.``assertHTMLEqual``` (*html1*, *html2*, *msg=None*)

断言字符串 `html1` 和 `html2` 是相等的。比较基于HTML语义。比较需要考虑以下因素：

- 忽略HTML标记之前和之后的空格。
- 所有类型的空格都被视为等同。
- 所有打开的标签被隐式地关闭，例如。当周围标签关闭或HTML文档结束时。
- 空标记等同于其自我关闭版本。
- HTML元素的属性顺序并不重要。
- 没有参数的属性等于名称和值相等的属性（参见示例）。

以下示例是有效的测试，不会引发任何 `AssertionError` ：

```
self.assertHTMLEqual('<p>Hello <b>world!</p>',
    '''<p>
        Hello   <b>world! <b/>
    </p>''')
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_t
erms" />',
    '<input id="id_accept_terms" type='checkbox' checked>'
)
```

`html1` 和 `html2` 必须是有效的HTML。如果其中一个无法解析，则会引发 `AssertionError` 。

出错时的输出可以用 `msg` 参数定制。

```SimpleTestCase.``assertHTMLNotEqual``` (*html1*, *html2*, *msg=None*)

认为字符串 `html1` 和 `html2` 的不是相等。比较基于HTML语义。有关详细信息，请参见 `assertHTMLEqual()` 。

`html1` 和 `html2` 必须是有效的HTML。如果其中一个无法解析，则会引发 `AssertionError` 。

出错时的输出可以用 `msg` 参数定制。

```SimpleTestCase.``assertXMLEqual``` (*xml1*, *xml2*, *msg=None*)
```

断言字符串 `xml1` 和 `xml2` 相等。比较基于XML语义。与 `assertHTMLEqual()` 类似，对解析的内容进行比较，因此仅考虑语义差异，而不考虑语法差异。当在任何参数中传递无效的XML时，始终会出现 `AssertionError` ，即使两个字符串都相同。

出错时的输出可以用 `msg` 参数定制。

`SimpleTestCase.``assertXMLNotEqual` (*xml1*, *xml2*, *msg=None*)

断言字符串 `xml1` 和 `xml2` 的不是相等。比较基于XML语义。有关详细信息，请参见 `assertXMLEqual()` 。

出错时的输出可以用 `msg` 参数定制。

`SimpleTestCase.``assertInHTML` (*needle*, *haystack*, *count=None*, *msg_prefix=''*)

断言HTML片段 `needle` 包含在 `haystack` 中。

如果指定 `count` 整数参数，则将严格验证 `needle` 出现的次数。

在大多数情况下，空白被忽略，属性排序不重要。传入的参数必须是有效的HTML。

`SimpleTestCase.``assertJSONEqual` (*raw*, *expected_data*, *msg=None*)

断言JSON片段 `raw` 和 `expected_data` 是相等的。通常的JSON非重要空格规则适用于将重量级委派给 `json` 库。

出错时的输出可以用 `msg` 参数定制。

`SimpleTestCase.``assertJSONNotEqual` (*raw*, *expected_data*, *msg=None*)

New in Django 1.8.

断言JSON片段 `raw` 和 `expected_data` 的不是相等。有关详细信息，请参见 `assertJSONEqual()` 。

出错时的输出可以用 `msg` 参数定制。

`TransactionTestCase.``assertQuerysetEqual` (*qs*, *values*, *transform=repr*, *ordered=True*, *msg=None*)

断言查询集 `qs` 返回值 `values` 。

使用函数 `transform` 执行 `qs` 和 `values` 的内容的比较；默认情况下，这意味着比较每个值的 `repr()` 。如果 `repr()` 未提供唯一或有帮助的比较，则可以使用任何其他可调用项。

默认情况下，比较也依赖于顺序。如果 `qs` 不提供隐式排序，可以将 `ordered` 参数设置为 `False` ，将比较结果转换为 `collections.` 计数器比较。如果顺序未定义（如果给定的 `qs` 不是有序的，并且比较是针对多个有序值），则会引

发 `ValueError` 。

出错时的输出可以用 `msg` 参数定制。

Changed in Django 1.7:

该方法现在接受 `msg`

`TransactionTestCase.``assertNumQueries` (*num*, *func*, **args*, ***kwargs*)

断言当使用 `*args` 和 `**kwargs` 调用 `func` 时，将执行 `num` 数据库查询。

如果 `kwargs` 中存在 `"using"` 键，它将用作要检查查询数的数据库别名。如果你想用 `using` 参数调用一个函数，你可以通过用 `lambda` 包装调用来添加一个额外的参数：

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

您还可以将其用作上下文管理器：

```
with self.assertNumQueries(2):
    Person.objects.create(name="Aaron")
    Person.objects.create(name="Daniel")
```

电子邮件服务

如果您的任何Django视图使用*Django's email functionality*发送电子邮件，您可能不希望在每次使用该视图运行测试时发送电子邮件。因此，Django的测试运行器会自动将所有Django发送的电子邮件重定向到一个虚拟发件箱。这使您可以测试发送电子邮件的每个方面 - 从发送到每封邮件内容的邮件数量，而不实际发送邮件。

测试运行器通过用测试后端透明地替换普通电子邮件后端来实现这一点。（不要担心，这对Django之外的任何其他电子邮件发件人（例如您计算机的邮件服务器）（如果您正在运行）没有任何影响。）

`django.core.mail.``outbox`

在测试运行期间，每封发出的电子邮件都保存在 `django.core.mail.outbox` 中。这是所有已发送的 `EmailMessage` 实例的简单列表。 `outbox` 属性是在使用 `locmem` 电子邮件后端时仅在创建的特殊属性。它通常不作为 `django.core.mail` 模块的一部分存在，您不能直接导入它。下面的代码显示了如何正确访问此属性。

下面是一个示例测试，检查 `django.core.mail.outbox` 的长度和内容：

```
from django.core import mail
from django.test import TestCase

class EmailTest(TestCase):
    def test_send_email(self):
        # Send message.
        mail.send_mail('Subject here', 'Here is the message.',
            'from@example.com', ['to@example.com'],
            fail_silently=False)

        # Test that one message has been sent.
        self.assertEqual(len(mail.outbox), 1)

        # Verify that the subject of the first message is correc
t.
        self.assertEqual(mail.outbox[0].subject, 'Subject here')
```

如前所述*previously*，测试发件箱在Django `*TestCase` 中的每个测试开始时都被清空。要手动清空发件箱，请将空列表分配给 `mail.outbox` ：

```
from django.core import mail

# Empty the test outbox
mail.outbox = []
```

管理命令

可以使用 `call_command()` 函数测试管理命令。输出可以重定向到 `StringIO` 实例：

```
from django.core.management import call_command
from django.test import TestCase
from django.utils.six import StringIO

class ClosepollTest(TestCase):
    def test_command_output(self):
        out = StringIO()
        call_command('closepoll', stdout=out)
        self.assertIn('Expected output', out.getvalue())
```

跳过测试

unittest库提供 `@skipIf` 和 `@skipUnless` 装饰器，如果提前知道这些测试在某些条件下会失败，您可以跳过测试。

例如，如果您的测试需要特定的可选库来成功，您可以使用 `@skipIf` 来装饰测试用例。然后，测试运行器将报告测试没有被执行以及为什么，而不是失败测试或完全省略测试。

为了补充这些测试跳过行为，Django提供了两个额外的跳过装饰器。这些装饰器不是测试通用布尔值，而是检查数据库的功能，如果数据库不支持特定的命名特性，则跳过测试。

装饰器使用字符串标识符来描述数据库特征。此字符串对应于数据库连接要素类的属性。有关可用作跳过测试的基础的数据库功能的完整列表，请参见 `django.db.backends.BaseDatabaseFeatures` 类。

`skipIfDBFeature` (*feature_name_strings*)

如果支持所有命名的数据库功能，请跳过装饰测试或 `TestCase` 。

例如，如果数据库支持事务（例如，不在PostgreSQL下运行，但在MySQL with MyISAM表下），则不会执行以下测试：

```
class MyTests(TestCase):
    @skipIfDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
```

Changed in Django 1.7:

`skipIfDBFeature` 现在可以用来装饰 `TestCase` 类。

Changed in Django 1.8:

`skipIfDBFeature` 可以接受多个要素字符串。

`skipUnlessDBFeature` (*feature_name_strings*)

如果任何命名的数据库功能不支持，请跳过装饰测试或 `TestCase` 。

例如，如果数据库支持事务（例如，它将在PostgreSQL下运行，但在MySQL和MyISAM表下的不是），则将执行以下测试：

```
class MyTests(TestCase):
    @skipUnlessDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
```

Changed in Django 1.7:

`skipUnlessDBFeature` 现在可用于装饰 `TestCase` 类。

Changed in Django 1.8:

`skipUnlessDBFeature` 可以接受多个要素字符串。

高级测试主题

请求工厂

class `RequestFactory`

`RequestFactory` 与测试客户端共享相同的API。但是，RequestFactory不是像浏览器一样工作，而是提供一种方法来生成可用作任何视图的第一个参数的请求实例。这意味着您可以像测试任何其他函数一样测试视图函数 - 作为一个黑盒子，具有完全已知的输入，测试特定的输出。

`RequestFactory` 的API是测试客户端API的一个稍微受限的子集：

- It only has access to the HTTP methods `get()` , `post()` , `put()` , `delete()` , `head()` , `options()` , and `trace()` .
- 这些方法接受所有相同的参数，除了 *`follows`* 的。因为这只是一个生产请求的工厂，所以由您来处理响应。
- 它不支持中间件。如果视图正常工作需要，会话和身份验证属性必须由测试本身提供。

例

以下是使用请求工厂的简单单元测试：

```
from django.contrib.auth.models import AnonymousUser, User
from django.test import TestCase, RequestFactory

from .views import my_view

class SimpleTest(TestCase):
    def setUp(self):
        # Every test needs access to the request factory.
        self.factory = RequestFactory()
        self.user = User.objects.create_user(
            username='jacob', email='jacob@…', password='top_sec
ret')

    def test_details(self):
        # Create an instance of a GET request.
        request = self.factory.get('/customer/details')

        # Recall that middleware are not supported. You can simu
late a
        # logged-in user by setting request.user manually.
        request.user = self.user

        # Or you can simulate an anonymous user by setting reque
st.user to
        # an AnonymousUser instance.
        request.user = AnonymousUser()

        # Test my_view() as if it were deployed at /customer/det
ails
        response = my_view(request)
        self.assertEqual(response.status_code, 200)
```

测试和多个数据库

测试主/副本配置

如果您使用主/副本（由某些数据库称为主/从属）测试多数据库配置，则创建测试数据库的这种策略会出现问题。创建测试数据库时，不会有任何复制，因此，在主节点上创建的数据将不会在副本上看到。

为了弥补这一点，Django允许您定义数据库是测试镜像。考虑以下（简化）示例数据库配置：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'myproject',
        'HOST': 'dbprimary',
         # ... plus some other settings
    },
    'replica': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'myproject',
        'HOST': 'dbreplica',
        'TEST_MIRROR': 'default'
        # ... plus some other settings
    }
}
```

In this setup, we have two database servers: `dbprimary`, described by the database alias `default`, and `dbreplica` described by the alias `replica`. 如您所料，`dbreplica` 已由数据库管理员配置为 `dbprimary` 的只读副本，因此在正常活动中，对 `default` 将显示在 `replica` 上。

如果Django创建了两个独立的测试数据库，这将破坏任何期望复制发生的测试。但是，`replica` 数据库已配置为测试镜像（使用 `TEST_MIRROR` 设置），表示在测试下，`replica` `default` 的镜像。

配置测试环境时，`replica` 的测试版本将不会创建。相反，到 `replica` 的连接将被重定向到指向 `default`。因此，对 `default` 的写入将出现在 `replica` 上-但是因为它们实际上是同一个数据库，而不是因为两个数据库之间存在数据复制。

控制测试数据库的创建顺序

默认情况下，Django将假定所有数据库都依赖于 `default` 数据库，因此，始终首先创建 `default` 数据库。但是，不能保证测试设置中任何其他数据库的创建顺序。

如果数据库配置需要特定的创建顺序，则可以使用 `TEST_DEPENDENCIES` 设置指定存在的依赖关系。考虑以下（简化）示例数据库配置：

```
DATABASES = {
    'default': {
         # ... db settings
         'TEST_DEPENDENCIES': ['diamonds']
    },
    'diamonds': {
        # ... db settings
         'TEST_DEPENDENCIES': []
    },
    'clubs': {
        # ... db settings
        'TEST_DEPENDENCIES': ['diamonds']
    },
    'spades': {
        # ... db settings
        'TEST_DEPENDENCIES': ['diamonds','hearts']
    },
    'hearts': {
        # ... db settings
        'TEST_DEPENDENCIES': ['diamonds','clubs']
    }
}
```

在此配置下，将首先创建 `diamonds` 数据库，因为它是唯一没有依赖关系的数据库别名。接下来将创建 `default` 和 `clubs` 别名（虽然不保证创建此对的顺序）；那么 `hearts` ；和最后 `spades` 。

如果在 `TEST_DEPENDENCIES` 定义中有任何循环依赖性，则会引发 `ImproperlyConfigured` 异常。

高级功能 `TransactionTestCase`

`TransactionTestCase.``available_apps`

警告

此属性是专用API。它可以在未来没有弃用期的情况下更改或删除，例如以适应应用程序加载的更改。

它用于优化Django自己的测试套件，其中包含数百个模型，但在不同应用程序中的模型之间没有关系。

默认情况下， `available_apps` 设置为 `None` 。每次测试后，Django调用 `flush` 以重置数据库状态。这将清空所有表并发出 `post_migrate` 信号，该信号会为每个模型重新创建一个内容类型和三个权限。此操作与模型的数量成比例地变得昂贵。

将 `available_apps` 设置为应用程序列表指示Django表现为只有这些应用程序中的模型可用。 `TransactionTestCase` 的行为更改如下：

- `post_migrate` 在每次测试之前触发，以便为可用应用中的每个模型创建内容类型和权限（如果缺少）。
- 每次测试后，Django只清空与可用应用程序中的模型对应的表。但是，在数据库级别，截断可能级联到不可用应用程序中的相关模型。此外，`post_migrate` 不会触发；在选择正确的应用程序集后，它将由下一个 `TransactionTestCase` 触发。

由于数据库未完全刷新，如果测试创建不包含在 `available_apps` 中的模型实例，则它们将泄漏，并且可能导致无关的测试失败。注意使用会话的测试；默认会话引擎将它们存储在数据库中。

由于冲洗数据库后不会发出 `post_migrate` ，因此在 `TransactionTestCase` 之后的状态与 `TestCase` 后的状态不同：由侦听器创建的行 `post_migrate` 。考虑*order in which tests are executed*，这不是问题，只要给定测试套件中的 `TransactionTestCase` 声明 `available_apps` 没有一个。

`available_apps` 在Django自己的测试套件中是必需的。

`TransactionTestCase.``reset_sequences`

在 `TransactionTestCase` 上设置 `reset_sequences = True测试运行：`

```
class TestsThatDependsOnPrimaryKeySequences(TransactionTestCase)
:
    reset_sequences = True

    def test_animal_pk(self):
        lion = Animal.objects.create(name="lion", sound="roar")
        # lion.pk is guaranteed to always be 1
        self.assertEqual(lion.pk, 1)
```

除非明确测试主键序列号，否则建议不要在测试中硬编码主键值。

使用 `reset_sequences = True` 会降低测试速度，因为主键重置是一个相对昂贵的数据库操作。

使用Django测试运行器测试可重用的应用程序

如果您正在编写*reusable application*，您可能需要使用Django测试运行器来运行自己的测试套件，从而从Django测试基础架构中受益。

通常的做法是应用程序代码旁边的*tests*目录，具有以下结构：

```
runtests.py
polls/
    __init__.py
    models.py
    ...
tests/
    __init__.py
    models.py
    test_settings.py
    tests.py
```

让我们来看看一些这些文件：

runtests.py

```
#!/usr/bin/env python
import os
import sys

import django
from django.conf import settings
from django.test.utils import get_runner

if __name__ == "__main__":
    os.environ['DJANGO_SETTINGS_MODULE'] = 'tests.test_settings'
    django.setup()
    TestRunner = get_runner(settings)
    test_runner = TestRunner()
    failures = test_runner.run_tests(["tests"])
    sys.exit(bool(failures))
```

这是您调用以运行测试套件的脚本。它设置Django环境，创建测试数据库并运行测试。

为了清楚起见，此示例仅包含使用Django测试运行器所需的最低限度。您可能需要添加用于控制详细程度的命令行选项，传递特定测试标签以运行等。

tests/test_settings.py

```
SECRET_KEY = 'fake-key'
INSTALLED_APPS = [
    "tests",
]
```

此文件包含运行应用测试所需的*Django settings*。

同样，这是一个最小的例子；您的测试可能需要额外的设置才能运行。

由于在运行测试时，测试包包含在 `INSTALLED_APPS` 中，因此您可以在其 `models.py` 文件中定义纯测试模型。

使用不同的测试框架

显然， `unittest` 不是唯一的Python测试框架。虽然Django不为替代框架提供显式支持，但它提供了一种方法来调用为替代框架构建的测试，就像他们是正常的Django测试一样。

当您运行 `./ manage.py 测试` 时，Django会查看 `TEST_RUNNER` 设置来确定要做什么。默认情况下， `TEST_RUNNER` 指向 `'django.test.runner.DiscoverRunner'` 。这个类定义了默认的Django测试行为。这种行为包括：

1. 执行全局预测试设置。
2. 在名称与模式 `test*.py` 匹配的当前目录下的任何文件中查找测试。
3. 创建测试数据库。
4. 运行 `migrate` 将模型和初始数据安装到测试数据库中。
5. 运行发现的测试。
6. 销毁测试数据库。
7. 执行全局后测试拆卸。

如果您在该类中定义了自己的测试运行器类并指向 `TEST_RUNNER` ，则Django将在您运行 `./ manage.py 测试时运行测试` 。这样，可以使用可以从Python代码执行的任何测试框架，或者修改Django测试执行过程以满足您可能遇到的任何测试需求。

定义测试运行器

测试运行器是定义 `run_tests()` 方法的类。Django附带一个定义默认Django测试行为的 `DiscoverRunner` 类。此类定义了 `run_tests()` 入口点，以及 `run_tests()` 用来设置，执行和删除测试套件的其他方法。

class `DiscoverRunner` (*pattern='test*.py'*, *top_level=None*, *verbosity=1*, *interactive=True*, *failfast=True*, *keepdb=False*, *reverse=False*, *debug_sql=False*, ***kwargs*)[source]

`DiscoverRunner` 将在符合 `pattern` 的任何文件中搜索测试。

`top_level` 可用于指定包含顶级Python模块的目录。通常Django可以自动计算出来，所以没有必要指定这个选项。如果指定，通常应该是包含您的 `manage.py` 文件的目录。

`verbosity` 确定将打印到控制台的通知和调试信息量； `0` 为无输出， `1` 为正常输出， `2` 为详细输出。

如果 `interactive` 是 `True` ，则测试套件有权在执行测试套件时询问用户指示。此行为的一个示例是请求删除现有测试数据库的权限。如果 `interactive` 是 `False` ，则测试套件必须能够在没有任何手动干预的情况下

运行。

如果 `failfast` 为 `True` ，则在检测到第一个测试失败后，测试套件将停止运行。

如果 `keepdb` 是 `True` ，测试套件将使用现有数据库，或者如果需要，创建一个。如果 `False` ，将创建一个新数据库，提示用户删除现有的数据（如果存在）。

如果 `reverse` 是 `True` ，测试用例将按相反的顺序执行。这可能有助于调试未正确隔离且具有副作用的测试。使用此选项时，将保留*Grouping by test class*。

如果 `debug_sql` 为 `True` ，失败的测试用例将输出记录到*django.db.backends logger*的SQL查询以及回溯。如果 `verbosity` 是 `2` ，则输出所有测试中的查询。

Django可以不时地通过添加新的参数来扩展测试运行器的能力。 `**kwargs` 声明允许此扩展。如果您子类化 `DiscoverRunner` 或编写自己的测试运行器，请确保接受 `**kwargs` 。

您的测试运行程序还可以定义其他命令行选项。Create or override an `add_arguments(cls, parser)` class method and add custom arguments by calling `parser.add_argument()` inside the method, so that the `test` command will be able to use those arguments.

Changed in Django 1.8:

以前，您必须向子类化测试运行器提供 `option_list` 属性，以向 `test` 命令可以使用的命令行选项列表中添加选项。

添加了 `keepdb` ， `reverse` 和 `debug_sql` 参数。

属性

`DiscoverRunner.``test_suite`

New in Django 1.7.

用于构建测试套件的类。默认情况下，它设置为 `unittest.` TestSuite。如果您希望实现不同的收集测试的逻辑，这可以被覆盖。

`DiscoverRunner.``test_runner`

New in Django 1.7.

这是用于执行单独测试和格式化结果的低级测试运行器的类。默认情况下，它设置为 `unittest.` TextTestRunner。尽管在命名约定中存在不幸的相似性，但这不是与 `DiscoverRunner` 相同类型的类，它涵盖了更广泛的职责。您可以覆盖此属性以修改运行和报告测试的方式。

`DiscoverRunner.``test_loader`

这是加载测试的类，无论是从TestCases或模块或其他方式，并将它们捆绑到测试套件中为跑步者执行。默认情况下，它设置为 `unittest.defaultTestLoader` 。如果您的测试将以不寻常的方式加载，您可以覆盖此属性。

`DiscoverRunner.``option_list`

这是 `optparse` 选项的元组，它将被送入管理命令的 `OptionParser` 以解析参数。有关更多详细信息，请参阅Python的 `optparse` 模块的文档。

自1.8版起已弃用：您现在应该覆盖 `add_arguments()` 类方法以添加由 `test` 管理命令接受的自定义参数。

方法

`DiscoverRunner.``run_tests` (*test_labels*, *extra_tests=None*, ***kwargs*) [source]

运行测试套件。

`test_labels` 允许您指定要运行的测试并支持多种格式（有关支持的格式的列表，请参阅 `DiscoverRunner.build_suite()` ）。

`extra_tests` 是要添加到测试运行程序执行的套件中的额外 `TestCase` 实例的列表。除了在 `test_labels` 中列出的模块中发现的那些测试之外，还运行这些额外的测试。

此方法应返回失败的测试数。

classmethod `DiscoverRunner.``add_arguments` (*parser*)[source]

New in Django 1.8.

覆盖此类方法以添加由 `test` 管理命令接受的自定义参数。请参见 `argparse.` ArgumentParser.add_argument()有关将参数添加到解析器的详细信息。

`DiscoverRunner.``setup_test_environment` (***kwargs*)[source]

通过调用 `setup_test_environment()` 并将 `DEBUG` 设置为 `False` 来设置测试环境。

`DiscoverRunner.``build_suite` (*test_labels*, *extra_tests=None*, ***kwargs*) [source]

构造与提供的测试标签匹配的测试套件。

`test_labels` 是描述要运行的测试的字符串列表。测试标签可以采用以下四种形式之一：

- `path.to.test_module.` TestCase.test_method - 在测试用例中运行单个测试方法。
- `path.to.test_module.` TestCase - 在测试用例中运行所有的测试方法。

- `path.to.module` - 在命名的Python包或模块中搜索并运行所有测试。
- `path/to/directory` - 搜索并运行命名目录下的所有测试。

如果 `test_labels` 的值为 `None` ，则测试运行器将在名称与其 `pattern` 匹配的当前目录下的所有文件中搜索测试以上）。

`extra_tests` 是要添加到测试运行程序执行的套件中的额外 `TestCase` 实例的列表。除了在 `test_labels` 中列出的模块中发现的那些测试之外，还运行这些额外的测试。

返回准备运行的 `TestSuite` 实例。

`DiscoverRunner.``setup_databases` (***kwargs*)[source]

创建测试数据库。

返回一个数据结构，提供足够的详细信息来撤销已做的更改。在测试结束时，此数据将提供给 `teardown_databases()` 函数。

`DiscoverRunner.``run_suite` (*suite*, ***kwargs*)[source]

运行测试套件。

返回运行测试套件所产生的结果。

`DiscoverRunner.``teardown_databases` (*old_config*, ***kwargs*)[source]

销毁测试数据库，恢复预测试条件。

`old_config` 是定义数据库配置中需要反转的更改的数据结构。它是 `setup_databases()` 方法的返回值。

`DiscoverRunner.``teardown_test_environment` (***kwargs*)[source]

恢复预测试环境。

`DiscoverRunner.``suite_result` (*suite*, *result*, ***kwargs*)[source]

计算并返回基于测试套件的返回码，以及该测试套件的结果。

测试实用程序

django.test.utils

为了帮助创建自己的测试运行器，Django在 `django.test.utils` 模块中提供了一些实用程序方法。

`setup_test_environment` ()[source]

执行任何全局预测试设置，例如安装模板呈现系统的设置和设置虚拟电子邮件发件箱。

`teardown_test_environment` ()[source]

执行任何全局后测试拆卸，例如删除模板系统中的黑魔法钩子，并恢复正常的电子邮件服务。

django.db.connection.creation

数据库后端的创建模块还提供了一些在测试期间可用的实用程序。

`create_test_db` ([*verbosity=1*, *autoclobber=False*, *serialize=True*, *keepdb=False*])

创建新的测试数据库，并对其运行 `migrate` 。

`verbosity` 具有与 `run_tests()` 中相同的行为。

`autoclobber` 描述了如果发现与测试数据库具有相同名称的数据库，则会发生的行为：

- 如果 `autoclobber` 是 `False` ，则会要求用户批准销毁现有数据库。如果用户不批准，则调用 `sys.exit` 。
- 如果autoclobber为 `True` ，则数据库将被销毁，而无需咨询用户。

`serialize` 确定Django在运行测试之前是否将数据库序列化为内存中的JSON字符串（用于在没有事务的情况下在测试之间恢复数据库状态）。如果您没有任何具有*serialized_rollback=True*的测试类，您可以将其设置为 `False` 以加快创建时间。

New in Django 1.7.1:

如果您使用默认测试运行器，则可以使用 `TEST` 字典中的 `SERIALIZE` 条目

`keepdb` 确定测试运行是否应使用现有数据库，或创建一个新的数据库。如果 `True` ，则将使用或创建现有数据库（如果不存在）。如果 `False` ，将创建一个新数据库，提示用户删除现有的数据（如果存在）。

返回它创建的测试数据库的名称。

`create_test_db()` 具有修改 `DATABASES` 中 `NAME` 的值的副作用，以匹配测试数据库的名称。

Changed in Django 1.7:

已添加 `serialize` 参数。

Changed in Django 1.8:

添加了 `keepdb` 参数。

`destroy_test_db` (*old_database_name*[, *verbosity=1*, *keepdb=False*])

销毁在 `DATABASES` 中名称为 `NAME` 的值的数据库，并将 `NAME` 设置为 `old_database_name` 的值。

`verbosity` 参数与 `DiscoverRunner` 具有相同的行为。

如果 `keepdb` 参数为 `True` ，则与数据库的连接将关闭，但数据库不会被销毁。

Changed in Django 1.8:

添加了 `keepdb` 参数。

与coverage.py集成

代码覆盖率描述了已经测试了多少源代码。它显示你的代码的哪些部分是由测试和哪些不是。它是测试应用程序的重要组成部分，因此强烈建议检查测试的覆盖率。

Django可以轻松地与coverage.py集成，这是一个用于测量Python程序代码覆盖率的工具。首先，安装coverage.py。接下来，从包含 `manage.py` 的项目文件夹中运行以下命令：

```
coverage run --source='.' manage.py test myapp
```

这将运行您的测试并收集项目中已执行文件的coverage数据。您可以通过键入以下命令查看此数据的报告：

```
coverage report
```

请注意，一些Django代码在运行测试时已执行，但由于传递给上一个命令的 `source` 标志，因此未在此处列出。

有关更多选项（如已注释的HTML列表，详细说明错过的行），请参阅coverage.py文档。

部署

部署 Django

虽然Django 满满的便捷性让Web 开发人员活得轻松一些，但是如果不能轻松地部署你的网站，这些工具还是没有什么用处。Django 起初，易于部署就是一个主要的目标。有许多优秀的方法可以轻松地来部署Django：

- 如何使用WSGI 部署
- 部署的检查清单

FastCGI 的支持已经废弃并将在Django 1.9 中删除。

- 如何使用FastCGI、SCGI 和AJP 部署Django

如果你是部署Django 和/或 Python 的新手，我们建议你先试试 mod_wsgi。 在大部分情况下，这将是最简单、最迅速和最稳当的部署选择。

> 另见
>
> Django Book（第二版）的第12 章 更详细地讨论了部署，尤其是可扩展性。但是请注意，这个版本是基于Django 1.1 版本编写，而且在 `mod_python` 废弃并于Django 1.5 中删除之后一直没有更新。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

如何使用WSGI 部署

Django 首要的部署平台是WSGI，它是Python Web 服务器和应用的标准。

Django 的 `startproject` 管理命名为你设置一个简单的默认WSGI 配置，你可以根据你项目的需要做调整并指定任何与WSGI 兼容的应用服务器使用。

Django 包含以下WSGI 服务器的入门文档：

- 如何使用Apache 和mod_wsgi 部署Django
- 从Apache 中利用Django 的用户数据库进行认证
- 如何使用Gunicorn 部署Django (100%)
- 如何使用uWSGI 部署Django (100%)

application 对象

使用WSGI 部署的核心概览是 `application` 可调用对象，应用服务器使用它来与你的代码进行交换。在Python 模块中，它通常一个名为 `application` 的对象提供给服务器使用。

`startproject` 命令创建一个 `<project_name>/wsgi.py` 文件，它就包含这样一个 `application` 可调用对象。

它既可用于Django 的开发服务器，也可以用于线上WSGI 的部署。

WSGI 服务器从它们的配置中获得 `application` 可调用对象的路径。Django 内建的服务器，叫做 `runserver` 和 `runfcgi` 命令，是从 `WSGI_APPLICATION` 设置中读取它。默认情况下，它设置为 `<project_name>.wsgi.application` ，指向 `<project_name>/wsgi.py` 中的 `application` 可调用对象。

配置settings 模块

当WSGI 服务器加载你的应用时，Django 需要导入settings 模块 —— 这里是你的全部应用定义的地方。

Django 使用 `DJANGO_SETTINGS_MODULE` 环境变量来定位settings 模块。它包含settings 模块的路径，以点分法表示。对于开发环境和线上环境，你可以使用不同的值；这完全取决于你如何组织你的settings。

如果这个变量没有设置，默认的 `wsgi.py` 设置为 `mysite.settings` ，其中 `mysite` 为你的项目的名称。这是 `runserver` 如何找到默认的 `settings` 文件的机制。

> 注
>
> 因为环境变量是进程范围的，当你在同一个进程中运行多个Django 站点时，它将不能工作。使用 `mod_wsgi` 就是这个情况。
>
> 为了避免这个问题，可以使用mod_wsgi 的守护进程模式，让每个站点位于它自己的守护进程中，或者在 `wsgi.py` 中通过强制使用 `os.environ["DJANGO_SETTINGS_MODULE"] = "mysite.settings"` 来覆盖这个值。

运用WSGI 中间件

你可以简单地封装application 对象来运用 WSGI 中间件。 例如，你可以在 `wsgi.py` 的底下添加以下这些行：

```
from helloworld.wsgi import HelloWorldApplication
application = HelloWorldApplication(application)
```

如果你结合使用 Django 的application 与另外一个WSGI application 框架，你还可以替换Django WSGI 的application 为一个自定义的WSGI application。

> 注
>
> 某些第三方的WSGI 中间件在处理完一个请求后不调用响应对象上的 `close` —— most notably Sentry's error reporting middleware up to version 2.0.7。这些情况下，不会发送 `request_finished` 信号。这可能导致数据库和memcache 服务的空闲连接。

> 译者：Django 文档协作翻译小组，原文：WSGI servers。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

如何使用Django与FastCGI，SCGI或AJP

自1.7版起已弃用：FastCGI支持已弃用，将在Django 1.9中删除。

虽然*WSGI*是Django的首选部署平台，但许多人使用共享托管，其中诸如FastCGI，SCGI或AJP等协议是唯一可行的选项。

注意

本文档主要关注FastCGI。还通过 `flup` Python包支持其他协议，如SCGI和AJP。有关SCGI和AJP的详细信息，请参阅下面的协议部分。

本质上，FastCGI是一种有效的方式，让外部应用程序将页面提供给Web服务器。Web服务器将传入的Web请求（通过套接字）委派给FastCGI，FastCGI执行代码并将响应传递回Web服务器，Web服务器反过来将其传递回客户端的Web浏览器。

像WSGI一样，FastCGI允许代码留在内存中，允许在没有启动时间的情况下提供请求。而例如。*mod_wsgi*可以嵌入Apache Web服务器进程或作为单独的守护进程配置，FastCGI进程永远不会在Web服务器进程内运行，始终在一个单独的持久进程中。

为什么在单独的进程中运行代码？

Apache中的传统 `mod_*` 安排将各种脚本语言（最值得注意的是PHP，Python和Perl）嵌入到Web服务器的进程空间中。虽然这会降低启动时间 - 因为代码不必为每个请求读取磁盘 - 它是以内存使用为代价的。

由于FastCGI的性质，甚至可能有不同于Web服务器进程的用户帐户运行的进程。这对共享系统是一个很好的安全好处，因为它意味着你可以保护你的代码与其他用户。

先决条件：flup

在开始使用FastCGI和Django之前，您需要安装flup，一个用于处理FastCGI的Python库。版本0.5或更高版本应该工作正常。

启动FastCGI服务器

FastCGI在客户端 - 服务器模型上运行，在大多数情况下，您将自己启动FastCGI进程。当服务器需要加载动态页面时，您的Web服务器（无论是Apache，lighttpd还是其他）只会联系您的Django-FastCGI进程。因为守护程序已经在内存中运行代码，所以它能够很快地提供响应。

注意

如果你在共享托管系统上，你可能会被迫使用Web服务器管理的FastCGI进程。有关更多信息，请参阅下面关于使用Web服务器管理的进程运行Django的部分。

Web服务器可以通过以下两种方式之一连接到FastCGI服务器：它可以使用Unix域套接字（Win32系统上的“命名管道”），也可以使用TCP套接字。你选择的是一种偏好的方式；TCP套接字通常更容易由于权限问题。

要启动您的服务器，首先切换到项目的目录（*manage.py*），然后运行 `runfcgi` 命令：

```
./manage.py runfcgi [options]
```

如果您在 `runfcgi` 之后指定 `help` 作为唯一选项，它将显示所有可用选项的列表。

您需要指定 `socket`，`protocol` 或 `host` 和 `port`。然后，当您设置Web服务器时，您只需将其指向在启动FastCGI服务器时指定的主机/端口或套接字。请参阅下面的示例。

协议

Django支持flup的所有协议，即fastcgi，SCGI和AJP1.3（Apache JServ协议，版本1.3）。通过使用 `protocol=<protocol_name>` 选项和 `./ manage.py runfcgi` 选择您的首选协议 - 其中 `<protocol_name>` 可以是以下之一：`fcgi`（默认值），`scgi` 或 `ajp`。例如：

```
./manage.py runfcgi protocol=scgi
```

例子

在TCP端口上运行线程服务器：

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

在Unix域套接字上运行预分拣的服务器：

```
./manage.py runfcgi method=prefork socket=/home/user/mysite.sock
 pidfile=django.pid
```

套接字安全

Django的默认umask要求web服务器和Django fastcgi进程使用同一组和用户运行。为了增加安全性，您可以在同一组下运行它们，但作为不同的用户。如果这样做，您需要使用 `runfcgi` 的 `umask` 参数将umask设置为0002。

运行没有守护进程（后台）的进程（有利于调试）：

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock maxrequests=1
```

停止FastCGI守护程序

如果您的进程在前台运行，可以轻松停止它：只需点击 `Ctrl-C` 将停止并退出FastCGI服务器。但是，当您处理后台进程时，您需要使用Unix `kill` 命令。

如果您为 `runfcgi` 指定 `pidfile` 选项，则可以杀死正在运行的FastCGI守护程序，如下所示：

```
kill `cat $PIDFILE`
```

...其中 `$PIDFILE` 是您指定的 `pidfile` 。

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env - \
  PYTHONPATH="../python:.." \
  ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

Apache设置

要在Apache和FastCGI上使用Django，您需要安装并配置Apache，并安装并启用mod_fastcgi。有关说明，请参阅Apache文档。

设置完成后，通过编辑 `httpd.conf` （Apache配置）文件将Apache指向Django FastCGI实例。你需要做两件事情：

- 使用 `FastCGIExternalServer` 指令指定FastCGI服务器的位置。
- 使用 `mod_rewrite` 可根据需要将网址指向FastCGI。

指定FastCGI服务器的位置

`FastCGIExternalServer` 指令告诉Apache如何找到您的FastCGI服务器。如FastCGIExternalServer docs说明，您可以指定 `socket` 或 `host` 。这里有两个例子：

```
# Connect to FastCGI via a socket / named pipe.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket
 /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 1
27.0.0.1:3033
```

在任一情况下，文件 `/home/user/public_html/mysite.fcgi` 实际上不必存在。（更多在此在下一节）。

使用mod_rewrite将URL指向FastCGI

第二步是告诉Apache对匹配某种模式的URL使用FastCGI。为此，请使用mod_rewrite模块并将URL重写到 `mysite.fcgi` （或您在 `FastCGIExternalServer` 指令中指定的任何内容，上一节）。

在本示例中，我们告诉Apache使用FastCGI来处理不表示文件系统上的文件并且不以 `/media/` 开头的任何请求。这可能是最常见的情况，如果你使用Django的管理网站：

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L,PT]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

在使用 `{% url %}` 构建网址时，Django会自动使用网址的预先重写版本。模板标签（和类似方法）。

使用mod_fcgid替代mod_fastcgi

通过FastCGI提供应用程序的另一种方法是使用Apache的mod_fcgid模块。与mod_fastcgi相比，mod_fcgid对FastCGI应用程序的处理方式不同，它管理工作进程本身的产生，不提供像 `FastCGIExternalServer` 这样的东西。这意味着配置看起来略有不同。

实际上，您必须像添加一个类似于稍后描述的在*shared-hosting environment*中运行Django的脚本处理程序。有关详细信息，请参阅mod_fcgid参考

lighttpd设置

lighttpd是一种通常用于提供静态文件的轻量级Web服务器。它本地支持FastCGI，因此，如果您的站点没有任何Apache特定的需求，它是服务静态页面和动态页面的不错选择。

请确保 `mod_fastcgi` 位于模块列表中 `mod_rewrite` 和 `mod_access` 之后的位置，而不是 `mod_accesslog` 之后。您可能也希望使用 `mod_alias` 来管理管理媒体。

将以下内容添加到lighttpd配置文件中：

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media" => "/home/user/django/contrib/admin/media/",
)

url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

在一个lighttpd上运行多个Django站点

lighttpd允许您使用“条件配置”允许每个主机自定义配置。要指定多个FastCGI站点，只需在每个站点的FastCGI配置周围添加一个条件块：

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
       ...
    )
    ...
}

# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
       ...
    )
    ...
}
```

您还可以通过在 `fastcgi.server` 指令中指定多个条目，在同一网站上运行多个Django安装。为每个添加一个FastCGI主机。

切诺基设置

Cherokee是一个非常快速，灵活和易于配置的Web服务器。它支持现在广泛的技术：FastCGI，SCGI，PHP，CGI，SSI，TLS和SSL加密连接，虚拟主机，验证，即时编码，负载平衡，Apache兼容日志文件，数据库平衡器，反向HTTP代理和多更多。

Cherokee项目提供了一个文档，用于与Cherokee一起设置Django。

在带有Apache的共享托管提供程序上运行Django

许多共享托管提供程序不允许您运行自己的服务器守护程序或编辑 `httpd.conf` 文件。在这些情况下，仍然可以使用Web服务器生成的进程运行Django。

注意

如果您正在使用Web服务器生成的进程，如本节所述，您无需自己启动FastCGI服务器。Apache将产生一些进程，按需扩展。

在Web根目录中，将其添加到名为 `.htaccess` 的文件：

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

然后，创建一个小脚本，告诉Apache如何生成您的FastCGI程序。创建一个文件 `mysite.fcgi` 并将其放在您的Web目录中，并确保使其可执行：

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

如果你的服务器使用mod_fastcgi这工作。另一方面，如果你正在使用mod_fcgid，除了在 `.htaccess` 文件中略有改动，设置基本相同。而不是添加一个fastcgi脚本处理程序，你必须添加一个fcgid-handler：

```
AddHandler fcgid-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

重新启动生成的服务器

如果您更改网站上的任何Python代码，您需要告诉FastCGI代码已更改。但在这种情况下不需要重新启动Apache。相反，只需重新上传 `mysite.fcgi` 或编辑文件，以便文件上的时间戳将更改。当Apache看到文件已更新时，它会为您重新启动Django应用程序。

如果您可以访问Unix系统上的命令shell，可以使用 `touch` 命令轻松完成此操作：

```
touch mysite.fcgi
```

投放管理媒体文件

无论您最终决定使用的服务器和配置，您还需要考虑如何提供管理媒体文件。在*mod_wsgi*文档中给出的建议也适用于上面详述的设置。

将URL前缀强制为特定值

因为许多这些基于fastcgi的解决方案需要在Web服务器的某个点重写URL，Django看到的路径信息可能不像传入的原始URL。这是一个问题，如果Django应用程序是从一个特定的前缀提供服务，你想要你的URL从 `{% url %}` 标记看起来像前缀，而不是重写的版本，其可能包含例如 `mysite.fcgi` 。

Django做一个很好的尝试来弄出真正的脚本名称前缀应该是什么。特别是，如果Web服务器设置 `SCRIPT_URL` （特定于Apache的mod_rewrite）或 `REDIRECT_URL` （由一些服务器设置，包括Apache + mod_rewrite在某些情况下）将自动处理原始前缀。

如果Django无法正确计算出前缀，以及您希望在网址中使用原始值，则可以在主要的 `settings` 文件中设置 `FORCE_SCRIPT_NAME` 设置。这将为通过该设置文件提供的每个网址统一设置脚本名称。因此，如果您希望不同的网址集在这种情况下具有不同的脚本名称，则需要使用不同的设置文件，但这是一种罕见的情况。

作为如何使用它的示例，如果您的Django配置为 `'/'` 下的所有URL提供服务，并且想要使用此设置，则可以设置 `FORCE_SCRIPT_NAME = ''` 。

部署静态文件

> 另见
>
> `django.contrib.staticfiles` 的用法简介，请参见管理静态文件（CSS、images）。

在线上环境部署静态文件

放置静态文件到线上环境的基本步骤很简单：当静态文件改变时，运行 `collectstatic` 命令，然后安排将收集好的静态文件的目录(`STATIC_ROOT`)搬到静态文件服务器上。取决于 `STATICFILES_STORAGE` ，这些文件可能需要手工移动到一个新的位置或者 `Storage` 类的 `post_process` 方法可以帮你。

当然，与所有的部署任务一样，魔鬼隐藏在细节中。每个线上环境的建立都会有所不同，所以你需要调整基本的纲要以适应你的需求。下面是一些常见的方法，可能有所帮助。

网站和静态文件位于同一台服务器上

如果你的静态文件和网站位于同一台服务器，流程可能像是这样：

- 将你的代码推送到部署的服务器上。
- 在这台服务器上，运行 `collectstatic` 来收集所有的静态文件到 `STATIC_ROOT` 。
- 配置Web 服务器来托管URL `STATIC_URL` 下的 `STATIC_ROOT` 。 例如，这是如何使用Apache 和mod_wsgi 来完成它。

你可能想自动化这个过程，特别是如果你有多台Web 服务器。有许多种方法来完成这个自动化，但是许多Django 开发人员喜欢 Fabric。

在一下的小节中，我们将演示一些示例的Fabric 脚本来自动化不同选择的文件部署。Fabric 脚本的语法相当简单，但这里不会讲述；参见Fabric 的文档 以获得其语法的完整解释。

所以，一个部署静态文件来多台Web 服务器上的Fabric 脚本大概会是：

```
from fabric.api import *

# Hosts to deploy onto
env.hosts = ['www1.example.com', 'www2.example.com']

# Where your project code lives on the server
env.project_root = '/home/www/myproject'

def deploy_static():
    with cd(env.project_root):
        run('./manage.py collectstatic -v0 --noinput')
```

静态文件位于一台专门的服务器上

大部分大型的Django 站点都使用一台单独的Web 服务器来存放静态文件 —— 例如一台不运行Django 的服务器。这种服务器通常运行一种不同类型的服务器 —— 更快但是功能很少。一些常见的选择有：

- Nginx
- 裁剪版的Apache

配置这些服务器在这篇文档范围之外；查看每种服务器各自的文档以获得说明。

既然你的静态文件服务器不会允许Django，你将需要修改的部署策略，大概会是这样：

- 当静态文件改变时，在本地运行 `collectstatic` 。
- 将你本地的 `STATIC_ROOT` 推送到静态文件服务器相应的目录中。在这一步，常见的选择rsync ，因为它只传输静态文件改变的部分。

下面是Fabric 脚本大概的样子：

```
from fabric.api import *
from fabric.contrib import project

# Where the static files get collected locally. Your STATIC_ROOT
 setting.
env.local_static_root = '/tmp/static'

# Where the static files should go remotely
env.remote_static_root = '/home/www/static.example.com'

@roles('static')
def deploy_static():
    local('./manage.py collectstatic')
    project.rsync_project(
        remote_dir = env.remote_static_root,
        local_dir = env.local_static_root,
        delete = True
    )
```

静态文件位于一个云服务或**CDN** 上

两位一个常见的策略是放置静态文档到一个云存储提供商比如亚马逊的S3 和/或一个CDN(Content Delivery Network)上。这让你可以忽略保存静态文件的问题，并且通常可以加快网页的加载（特别是使用CDN 的时候）。

当使用这些服务时，除了不是使用rsync 传输你的静态文件到服务器上而是到存储提供商或CDN 上之外，基本的工作流程和上面的差不多。

有许多方式可以实现它，但是如果提供商具有API，那么自定义的文件存储后端 将使得这个过程相当简单。如果你已经写好或者正在使用第三方的自定义存储后端，你可以通过设置 `STATICFILES_STORAGE` 来告诉 `collectstatic` 来使用它。

例如，如果你已经在 `myproject.storage.S3Storage` 中写好一个S3 存储的后端，你可以这样使用它：

```
STATICFILES_STORAGE = 'myproject.storage.S3Storage'
```

一旦完成这个，你所要做的就是运行 `collectstatic` ，然后你的静态文件将被你的存储后端推送到S3 上。如果以后你需要切换到一个不同的存储提供商，你只需简单地修改你的 `STATICFILES_STORAGE` 设置。

关于如何编写这些后端的细节，请参见编写一个自定义的存储系统。有第三方的应用提供存储后端，它们支持许多常见的文件存储API。一个不错的入口是 djangopackages.com 的概览 。

了解更多

关于 `django.contrib.staticfiles` 中包含的设置、命令、模板标签和其它细节，参见staticfiles 参考。

> 译者：Django 文档协作翻译小组，原文：Deploying static files。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

错误报告

当你运行一个公开站点时，你应该始终关闭 `DEBUG` 设置。这会使你的服务器运行得更快，也会防止恶意用户看到由错误页面展示的一些应用细节。

但是，运行在 `DEBUG` 为 `False` 的情况下，你不会看到你的站点所生成的错误 -- 每个人都只能看到公开的错误页面。你需要跟踪部署的站点上的错误，所以可以配置Django来生成带有错误细节的报告。

报告邮件

服务器错误

`DEBUG` 为 `False` 的时候，无论什么时候代码产生了未处理的异常，并且出现了服务器内部错误（HTTP状态码 500），Django 都会给 `ADMINS` 设置中的用户发送邮件。 这会向管理员提供任何错误的及时通知。 `ADMINS` 会得到一份错误的描述，完整的Python traceback，以及HTTP请求和导致错误的详细信息。

注意

为了发送邮件，DJango需要一些设置来告诉它如何连接到邮件服务器。最起码，你需要指定 `EMAIL_HOST` ，可能需要 `EMAIL_HOST_USER` 和 `EMAIL_HOST_PASSWORD` ，尽管所需的其他设置可能也依赖于你的邮件服务器的配置。邮件相关设置的完整列表请见 *Django设置文档*。

Django通常从root@localhost发送邮件。但是一些邮件提供商会拒收所有来自这个地址的邮件。修改 `SERVER_EMAIL` 设置可以使用不同的发信人地址。

将收信人的邮箱地址放入 `ADMINS` 设置中来激活这一行为。

另见

服务器错误邮件使用日志框架来发送，所以你可以通过 自定义你的日志配置自定义这一行为。

404错误

也可以配置Django来发送关于死链的邮件（404"找不到页面"错误）。Django在以下情况发送404错误的邮件：

- `DEBUG` 为 `False` ；
- 你的 `MIDDLEWARE_CLASSES` 设置含有 `django.middleware.common.BrokenLinkEmailsMiddleware` 。

如果符合这些条件，无论什么时候你的代码产生404错误，并且请求带有referer，Django 都会给 `MANAGERS` 中的用户发送邮件。 (It doesn't bother to email for 404s that don't have a referer – those are usually just people typing in broken URLs or broken Web 'bots).

注意

`BrokenLinkEmailsMiddleware` 必须出现在其它拦截404错误的中间件之前，比如 `LocaleMiddleware` 或者 `FlatpageFallbackMiddleware` 。把它放在你的 `MIDDLEWARE_CLASSES` 设置的最上面。

你可以通过调整 `IGNORABLE_404_URLS` 设置，告诉Django停止报告特定的404错误。它应该为一个元组，含有编译后的正则表达式对象。例如：

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'\.(php|cgi)$'),
    re.compile(r'^/phpmyadmin/'),
)
```

在这个例子中，任何以 `.php` 或者 `.cgi` 结尾URL的404错误都不会报告。任何以 `/phpmyadmin/` 开头的URL也不会。

下面的例子展示了如何排除一些浏览器或爬虫经常请求的常用URL：

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'^/apple-touch-icon.*\.png$'),
    re.compile(r'^/favicon\.ico$'),
    re.compile(r'^/robots\.txt$'),
)
```

（要注意这些是正则表达式，所以需要在句号前面添加反斜线来对它转义。）

如果你打算进一步自定义 `django.middleware.common.BrokenLinkEmailsMiddleware` 的行为（比如忽略来自web爬虫的请求），你应该继承它并覆写它的方法。

另见

404错误使用日志框架来记录。通常，日志记录会被忽略，但是你可以通过编写合适的处理器和配置日志，将它们用于错误报告。

过滤错误报告

过滤敏感的信息

错误报告对错误的调试及其有用，所以对于这些错误，通常它会尽可能多的记录下相关信息。例如，通常DJango会为产生的异常记录完整的traceback，traceback 帧的每个局部变量，以及 `HttpRequest` 的属性。

然而，有时特定的消息类型十分敏感，并不适合跟踪消息，比如用户的密码或者信用卡卡号。所以Django提供一套函数装饰器，来帮助你控制需要在生产环境（也就是 `DEBUG` 为 `False` 的情况）中的错误报告中过滤的消息：`sensitive_variables()` 和 `sensitive_post_parameters()` 。

`sensitive_variables` (*variables)[source]

如果你的代码中一个函数（视图或者常规的回调）使用可能含有敏感信息的局部变量，你可能需要使用 `sensitive_variables` 装饰器，来阻止错误报告包含这些变量的值。

```
from django.views.decorators.debug import sensitive_variables

@sensitive_variables('user', 'pw', 'cc')
def process_info(user):
    pw = user.pass_word
    cc = user.credit_card_number
    name = user.name
    ...
```

在上面的例子中， `user` ，`pw` 和 `cc` 变量的值会在错误报告中隐藏并且使用星号(**) 来代替，虽然 `name` 变量的值会公开。

要想有顺序地在错误报告中隐藏一个函数的所有局部变量，不要向 `sensitive_variables` 装饰器提供任何参数：

```
@sensitive_variables()
def my_function():
    ...
```

使用多个装饰器的时候

如果你想要隐藏的变量也是一个函数的参数（例如，下面例子中的 `user` ），并且被装饰的函数有多个装饰器，你需要确保将 `@sensitive_variables` 放在装饰器链的顶端。这种方法也会隐藏函数参数，尽管它通过其它装饰器传递：

```
@sensitive_variables('user', 'pw', 'cc')
@some_decorator
@another_decorator
def process_info(user):
    ...
```

`sensitive_post_parameters` (*parameters)[source]

如果你的代码中一个视图接收到了可能带有敏感信息的，带有 POST 参数 的 HttpRequest 对象，你可能需要使用 sensitive_post_parameters 装饰器，来阻止错误报告包含这些参数的值。

```
from django.views.decorators.debug import sensitive_post_paramet
ers

@sensitive_post_parameters('pass_word', 'credit_card_number')
def record_user_profile(request):
    UserProfile.create(user=request.user,
                       password=request.POST['pass_word'],
                       credit_card=request.POST['credit_card_num
ber'],
                       name=request.POST['name'])
    ...
```

在上面的例子中， pass_word 和 credit_card_number POST参数的值会在错误报告中隐藏并且使用星号(**) 来代替，虽然 name 变量的值会公开。

要想有顺序地在错误报告中隐藏一个请求的所有POST 参数，不要向 sensitive_post_parameters 装饰器提供任何参数：

```
@sensitive_post_parameters()
def my_view(request):
    ...
```

所有POST参数按顺序被过滤出特定 django.contrib.auth.views 视图的错误报告（ login ， password_reset_confirm ， password_change ， add_view 和 auth 中的 user_change_password ），来防止像是用户密码这样的敏感信息的泄露。

自定义错误报告

所有 sensitive_variables() 和 sensitive_post_parameters() 分别用敏感变量的名字向被装饰的函数添加注解，以及用POST敏感参数的名字向 HttpRequest 对象添加注解，以便在错误产生时可以随后过滤掉报告中的敏感信息。Django的默认错误包告过滤器 django.views.debug.SafeExceptionReporterFilter 会完成实际的过滤操作。 产生错误报告的时候，这个过滤器使用装饰器的注解来将相应的值替换为星号 (**) 。如果你希望为你的整个站点覆写或自定义这一默认的属性，你需要定义你自己的过滤器类，并且通过 DEFAULT_EXCEPTION_REPORTER_FILTER 设置来让Django使用它。

```
DEFAULT_EXCEPTION_REPORTER_FILTER = 'path.to.your.CustomExceptio
nReporterFilter'
```

你也可能会以更精细的方式来控制在提供的视图中使用哪种过滤器，通过设置 `HttpRequest` 的 `exception_reporter_filter` 属性。

```
def my_view(request):
    if request.user.is_authenticated():
        request.exception_reporter_filter = CustomExceptionRepor
terFilter()
    ...
```

你的自定义过滤器类需要继承自 `django.views.debug.SafeExceptionReporterFilter` ，并且可能需要覆写以下方法：

class `SafeExceptionReporterFilter` [source]

`SafeExceptionReporterFilter.``is_active` (*request*)[source]

如果其它方法中操作的过滤器已激活，返回 `True` 。如果 `DEBUG` 为 `False` ，通常过滤器是激活的。

`SafeExceptionReporterFilter.``get_request_repr` (*request*)

Returns the representation string of the request object, that is, the value that would be returned by `repr(request)` , except it uses the filtered dictionary of POST parameters as determined by `SafeExceptionReporterFilter.get_post_parameters()` .

`SafeExceptionReporterFilter.``get_post_parameters` (*request*)[source]

返回过滤后的POST参数字典。通常它会把敏感参数的值以星号 (**)替换。

`SafeExceptionReporterFilter.``get_traceback_frame_variables` (*request*, *tb_frame*)[source]

返回过滤后的，所提供traceback帧的局部变量的字典。通常它会把敏感变量的值以星号 (**)替换。

另见

你也可以通过编写自定义的*exception middleware*来建立自定义的错误报告。如果你编写了自定义的错误处理器，模拟Django内建的错误处理器，只在 `DEBUG` 为 `False` 时报告或记录错误是个好主意。

> 译者：Django 文档协作翻译小组，原文：Tracking code errors by email。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Admin

Django 最受欢迎的特性之一 —— 自动生成的Admin 界面的所有内容：

Django Admin 站点

Django 最强大的部分之一是自动生成的Admin 界面。它读取模型中的元数据来提供一个强大的、生产环境就绪的界面，使内容提供者能立即用它向站点中添加内容。在这篇文档中，我们讨论如何去激活、使用和自定义Django 的Admin 界面。

概述

通过使用 `startproject` 创建的默认项目模版中，Admin 已启用。

下面的一些要求作为参考：

1. 添加 `'django.contrib.admin'` 到 `INSTALLED_APPS` 设置中.
2. admin有四个依赖 - `django.contrib.auth` , `django.contrib.contenttypes` , `django.contrib.messages` 和 `django.contrib.sessions` . 如果这些应用没有在 `INSTALLED_APPS` 列表中, 那你要把它们添加到该列表中.
3. 把 `django.contrib.messages.context_processors.messages` 添加到 `TEMPLATES` 中 `DjangoTemplates` 后台的 `'context_processors'` 选项中，同样把 `django.contrib.auth.middleware.AuthenticationMiddleware` 和 `django.contrib.messages.middleware.MessageMiddleware` 添加到 `MIDDLEWARE_CLASSES` . (这些默认都是激活的，所以如果你手工操作过的话就需要按照以上方法进行设置..)
4. 确定应用中的哪些模型应该在Admin 界面中可以编辑。
5. 给每个模型创建一个 `ModelAdmin` 类，封装模型自定义的Admin 功能和选项。
6. 实例化 `AdminSite` 并且告诉它你的每一个模块和 `ModelAdmin` 类.
7. 将 `AdminSite` 实例绑定到URLconf。

做了这些步骤之后, 你将能够通过你已经绑定的URL来访问Django管理站点(默认是 `/admin/`).

其他话题

- Admin 行为
- Django admin 文档生成

注意

如何在产品中使用admin相关的静态文件(图片,JavaScript和CSS)的办法, 请参阅文件服务

还有什么问题? 试试 *FAQ: The admin*.

ModelAdmin objects

class `ModelAdmin`

`ModelAdmin` 类是模型在Admin 界面中的表示形式。通常，将它们在你的应用中的名为 `admin.py` 的文件里。让我们来看一个关于 `ModelAdmin` 类非常简单的例子:

```
from django.contrib import admin
from myproject.myapp.models import Author

class AuthorAdmin(admin.ModelAdmin):
    pass
admin.site.register(Author, AuthorAdmin)
```

你真的需要一个 `ModelAdmin` 对象吗?

在上面的例子中， `ModelAdmin` 并没有定义任何自定义的值。因此, 系统将使用默认的Admin 界面。如果对于默认的Admin 界面足够满意，那你根本不需要自己定义 `ModelAdmin` 对象, 你可以直接注册模型类而无需提供 `ModelAdmin` 的描述。那么上面的例子可以简化成：

```
from django.contrib import admin
from myproject.myapp.models import Author

admin.site.register(Author)
```

注册装饰器

`register` (**models*[, *site=django.admin.sites.site*])

New in Django 1.7.

还可以用一个装饰来注册您的 `ModelAdmin` 类（这里有关装饰器的详细信息,请参考python中的相关说明）:

```
from django.contrib import admin
from .models import Author

@admin.register(Author)
class AuthorAdmin(admin.ModelAdmin):
    pass
```

如果不使用默认的 `AdminSite` ，可以提供一个或多个模块类来注册 `ModelAdmin` 并且一个选择性关键参数 `site` （这里使用装饰器来注册需要注册的类和模块的，请特别留意紧跟装饰器后面关于ModelAdmin的声明，前面是Author，后面是

PersonAdmin，我的理解是后一种情况 下注册的类都可以用PersonAdmin来作为接口）：

```
from django.contrib import admin
from .models import Author, Reader, Editor
from myproject.admin_site import custom_admin_site

@admin.register(Author, Reader, Editor, site=custom_admin_site)
class PersonAdmin(admin.ModelAdmin):
    pass
```

发现admin 文件

当你将 `'django.contrib.admin'` 加入到 `INSTALLED_APPS` 设置中, Django就会自动搜索每个应用的 `admin` 模块并将其导入。

class `apps.``AdminConfig`

New in Django 1.7.

这是 admin的默认 `AppConfig` 类. 它在 Django 启动时调用 `autodiscover()` .

class `apps.``SimpleAdminConfig`

New in Django 1.7.

这个类和 `AdminConfig` 的作用一样,除了它不调用 `autodiscover()` .

`autodiscover` ()[source]

这个函数尝试导入每个安装的应用中的 `admin` 模块。这些模块用于注册模型到 Admin 中。

Changed in Django 1.7:

以前的Django版本推荐直接在URLconf中调用这个函数. Django 1.7不再需要这样. `AdminConfig` 能够自动的运行 auto-discovery.

如果你使用自定义的 `AdminSite` , 一般是导入所有的 `ModelAdmin` 子类到你的代码中并将其注册到自定义的 `AdminSite` 中. 在这种情况下， 为了禁用auto-discovery,在你的 `INSTALLED_APPS` 设置中，应该用 `'django.contrib.admin.apps.SimpleAdminConfig'` 代替 `'django.contrib.admin'` 。

Changed in Django 1.7:

在以前的版本中,admin需要被指示寻找 `admin.py` 文件通过 `autodiscover()` . 在Django 1.7, auto-discovery默认可用的，必须明确的使它失效当不需要时.

ModelAdmin options

`ModelAdmin` 非常灵活。 它有几个选项来处理自定义界面。 所有的选项都在 `ModelAdmin` 子类中定义：

```
from django.contrib import admin

class AuthorAdmin(admin.ModelAdmin):
    date_hierarchy = 'pub_date'
```

`ModelAdmin.``actions`

在修改列表页面可用的操作列表。详细信息请查看*Admin actions* .

`ModelAdmin.``actions_on_top`

`ModelAdmin.``actions_on_bottom`

控制actions bar 出现在页面的位置。默认情况下，admin的更改列表将操作显示在页面的顶部(`actions_on_top = True;` actions_on_bottom=False）。

`ModelAdmin.``actions_selection_counter`

控制选择计数器是否紧挨着下拉菜单action默认的admin 更改列表将会显示它 (`actions_selection_counter = True`).

`ModelAdmin.``date_hierarchy`

把 `date_hierarchy` 设置为在你的model 中的 `DateField` 或 `DateTimeField` 的字段名，然后更改列表将包含一个依据这个字段基于日期的下拉导航。

例如:

```
date_hierarchy = 'pub_date'
```

这将根据现有数据智能地填充自己，例如，如果所有的数据都是一个月里的, 它将只显示天级别的数据.

注意

`date_hierarchy` 在内部使用 `QuerySet.datetimes()` . 当时区支持启用时，请参考它的一些文档说明。(`USE_TZ = True`).

`ModelAdmin.``exclude`

如果设置了这个属性，它表示应该从表单中去掉的字段列表。

例如，让我们来考虑下面的模型：

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3)
    birth_date = models.DateField(blank=True, null=True)
```

如果你希望 `Author` 模型的表单只包含 `name` 和 `title` 字段, 你应该显式说明 `fields` 或 `exclude` ，像这样：

```
from django.contrib import admin

class AuthorAdmin(admin.ModelAdmin):
    fields = ('name', 'title')

class AuthorAdmin(admin.ModelAdmin):
    exclude = ('birth_date',)
```

由于Author 模型只有三个字段， `name` 、 `title` 和 `birth_date` ，上述声明产生的表单将包含完全相同的字段。

`ModelAdmin.``fields`

如果需要实现字段的布局中的“添加” 和 “更改”，“更改”网页形式的简单改变像只显示可用字段的一个子集，你可以使用 `fields` 选项修改他们的顺序或者行内分组(需要复杂布局的请参阅 `fieldsets` 选项将在下一段讲到). 例如,可以定义一个简单的管理表单的版本使用 `django.contrib.flatpages.models.FlatPage` 模块像下面这样:

```
class FlatPageAdmin(admin.ModelAdmin):
    fields = ('url', 'title', 'content')
```

在上面的例子中, 只有字段 `url` , `title` 和 `content` 将会在表单中顺序的显示. `fields` 能够包含在 `ModelAdmin.readonly_fields` 中定义的作为只读显示的值

不同于 `list_display` ， `fields` 选项 只包含model中的字段名或者通过 `form` 指定的表单。只有当它们列在 `readonly_fields` 中，它才能包含callables

要在同一行显示多个字段， 就把那些字段打包在一个元组里。例子中， `url` 和 `title` 字段 会显示在同一行， `content` 字段将会显示在他们的下一行里：

```
class FlatPageAdmin(admin.ModelAdmin):
    fields = (('url', 'title'), 'content')
```

注意

此 `字段` 选项不应与 `fieldsets` 选项中的 `fields` 字典键混淆，如下一节所述。

如果 `fields` 和 `fieldsets` 选项都不存在, Django将会默认显示每一个不是 `AutoField` 并且 `editable=True` 的字段, 在单一的字段集，和在模块中定义的字段有相同的顺序

ModelAdmin.``fieldsets

设置 `fieldsets` 控制管理“添加”和 “更改” 页面的布局.

`fieldsets` 是一个以二元元组为元素的列表, 每一个二元元组代表一个在管理表单的 `<fieldset>` (`<fieldset>` 是表单的一部分.)

二元元组的格式是 `(name, field_options)` , 其中 `name` 是一个字符串相当于fieldset的标题， `field_options` 是一个关于 fieldset的字典信息,一个字段列表包含在里面。

一个完整的例子, 来自于 `django.contrib.flatpages.models.FlatPage` 模块:

```
from django.contrib import admin

class FlatPageAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('url', 'title', 'content', 'sites')
        }),
        ('Advanced options', {
            'classes': ('collapse',),
            'fields': ('enable_comments', 'registration_required
', 'template_name')
        }),
    )
```

在管理界面的结果看起来像这样:

URL:
Example: "/about/contact/". Make sure to have leading and trailing slashes.
Title:
Content:
Full HTML is allowed.
Sites:
blogs.ljworld.com
internal.ljworld.com
www2.kusports.com
www2.ljworld.com
www.6newslawrence.com
www.6productions.com
www.lawrence.com
Hold down "Control", or "Command" on a Mac, to select more than one.
Advanced options
Show Advanced options
Save and add another | Save and continue editing | Save

如果 `fields` 和 `fieldsets` 选项都不存在, Django将会默认显示每一个不是 `AutoField` 并且 `editable=True` 的字段, 在单一的字段集，和在模块中定义的字段有相同的顺序。

`field_options` 字典有以下关键字:

- `fields`

  字段名元组将显示在该fieldset. 此键必选.

  例如:

  ```
  {
  'fields': ('first_name', 'last_name', 'address', 'city', 'state'),
  }
  ```

  就像 `fields` 选项, 显示多个字段在同一行, 包裹这些字段在一个元组. 在这个例子中, `first_name` 和 `last_name` 字段将显示在同一行:

  ```
  {
  'fields': (('first_name', 'last_name'), 'address', 'city', 'state'),
  }
  ```

  `fields` 能够包含定义在 `readonly_fields` 中显示的值作为只读.

如果添加可调用的名称到 `fields` 中,相同的规则适用于 `fields` 选项: 可调用的必须在 `readonly_fields` 列表中.

- `classes`

  一个列表包含额外的CSS classes 应用到 fieldset.

  例如:

  ```
  {
  'classes': ('wide', 'extrapretty'),
  }
  ```

  通过默认的管理站点样式表定义的两个有用的classes 是 `collapse` 和 `wide` . Fieldsets 使用 `collapse` 样式将会在初始化时展开并且替换掉一个 “click to expand” 链接. Fieldsets 使用 `wide` 样式将会有额外的水平空格.

- `description`

  一个可选择额外文本的字符串显示在每一个fieldset的顶部,在fieldset头部的底下. 字符串没有被 `TabularInline` 渲染由于它的布局.

  记住这个值不是 HTML-escaped 当它显示在管理接口中时. 如果你愿意，这允许你包括HTML。另外，你可以使用纯文本和 `django.utils.html.escape()` 避免任何HTML特殊字符。

`ModelAdmin.``filter_horizontal`

默认的, `ManyToManyField` 会在管理站点上显示一个 `<select multiple>` .（多选框）．但是，当选择多个时多选框非常难用. 添加一个 `ManyToManyField` 到该列表将使用一个漂亮的低调的JavaScript中的“过滤器”界面,允许搜索选项。选和不选选项框并排出现。参考 `filter_vertical` 使用垂直界面。

`ModelAdmin.``filter_vertical`

与 `filter_horizontal` 相同，但使用过滤器界面的垂直显示，其中出现在所选选项框上方的未选定选项框。

`ModelAdmin.``form`

默认情况下， 会根据你的模型动态创建一个 `ModelForm` 。 它被用来创建呈现在添加/更改页面上的表单。你可以很容易的提供自己的 `ModelForm` 来重写表单默认的添加/修改行为。或者，你可以使用 `ModelAdmin.get_form()` 方法自定义默认的表单，而不用指定一个全新的表单。

例子见添加自定义验证到*Admin* 中部分。

注

如果你在 ModelForm 中定义 `Meta.model` 属性，那么也必须定义 `Meta.fields` 或 `Meta.exclude` 属性。然而，当admin本身定义了fields，则 `Meta.fields` 属性将被忽略。

如果 `ModelForm` 仅仅只是给Admin 使用，那么最简单的解决方法就是忽略 `Meta.model` 属性，因为 `ModelAdmin` 将自动选择应该使用的模型。或者，你也可以设置在 `Meta` 类中的 `fields = []` 来满足 `ModelForm` 的合法性。

提示

如果 `ModelForm` 和 `ModelAdmin` 同时定义了一个 `exclude` 选项，那么 `ModelAdmin` 具有更高的优先级：

```
from django import forms
from django.contrib import admin
from myapp.models import Person

class PersonForm(forms.ModelForm):

    class Meta:
        model = Person
        exclude = ['name']

class PersonAdmin(admin.ModelAdmin):
    exclude = ['age']
    form = PersonForm
```

在上例中，“age” 字段将被排除而 “name” 字段将被包含在最终产生的表单中。

ModelAdmin.``formfield_overrides

这个属性通过一种临时的方案来覆盖现有的模型中 Field （字段）类型在admin site中的显示类型。 `formfield_overrides` 在类初始化的时候通过一个字典类型的变量来对应模型字段类型与实际重载类型的关系。

因为概念有点抽象，所以让我们来举一个具体的例子。 `formfield_overrides` 常被用于让一个已有的字段显示为自定义控件。所以，试想一下我们写了一个 RichTextEditorWidget（富文本控件） 然后我们想用它来代替 <textarea>（文本域控件） 用于输入大段文字。下面就是我们如何做到这样的替换。

```
from django.db import models
from django.contrib import admin

# Import our custom widget and our model from where they're defi
ned
from myapp.widgets import RichTextEditorWidget
from myapp.models import MyModel

class MyModelAdmin(admin.ModelAdmin):
    formfield_overrides = {
        models.TextField: {'widget': RichTextEditorWidget},
    }
```

注意字典的键是一个实际的字段类型，而不是一个具体的字符。 字典的值是另外一个字典结构的数据；这个参数会传递到表单字段 `__init__()`（初始化方法） 中。有关详细信息，请参见*The Forms API*。

警告

如果你想用一个关系字段的自定义界面(即 `ForeignKey` 或者 `ManyToManyField`)， 确保你没有在 `raw_id_fields` or `radio_fields` 中included那个字段名。

`formfield_overrides` 不会让您更改 `raw_id_fields` 或 `radio_fields` 设置的关系字段上的窗口小部件。这是因为 `raw_id_fields` 和 `radio_fields` 暗示自己的自定义小部件。

`ModelAdmin.``inlines`

请参见下面的 `InlineModelAdmin` 对象以及 `ModelAdmin.get_formsets_with_inlines()` 。

`ModelAdmin.``list_display`

使用 `list_display` 去控制哪些字段会显示在Admin 的修改列表页面中。

示例：

```
list_display = ('first_name', 'last_name')
```

如果你没有设置 `list_display` ，Admin 站点将只显示一列表示每个对象的 `__str__()` （Python 2 中是 `__unicode__()` ）。

在 `list_display` 中，你有4种赋值方式可以使用：

- 模型的字段。 例如:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name')
```

- 一个接受对象实例作为参数的可调用对象。例子：

```
def upper_case_name(obj):
    return ("%s  %s" % (obj.first_name, obj.last_name)).uppe
r()
upper_case_name.short_description = 'Name'

class PersonAdmin(admin.ModelAdmin):
    list_display = (upper_case_name,)
```

- 一个表示 `ModelAdmin` 中某个属性的字符串。行为与可调用对象相同。 例如：

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('upper_case_name',)

    def upper_case_name(self, obj):
        return ("%s  %s" % (obj.first_name, obj.last_name)).
upper()
    upper_case_name.short_description = 'Name'
```

- 表示模型中某个属性的字符串。它的行为与可调用对象几乎相同，但这时的 `self` 是模型实例。这里是一个完整的模型示例：

```
from django.db import models
from django.contrib import admin

class Person(models.Model):
    name = models.CharField(max_length=50)
    birthday = models.DateField()

    def decade_born_in(self):
        return self.birthday.strftime('%Y')[:3] + "0's"
    decade_born_in.short_description = 'Birth decade'

class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'decade_born_in')
```

关于 `list_display` 要注意的几个特殊情况：

- 如果字段是一个 `ForeignKey`，Django 将展示相关对象的 `__str__()`（Python 2 上是 `__unicode__()`）。

- 不支持 `ManyToManyField` 字段，因为这将意味着对表中的每一行执行单独的SQL 语句。如果尽管如此你仍然想要这样做，请给你的模型一个自定义的方法，并将该方法名称添加到 `list_display` 。（ `list_display` 的更多自定义方法请参见下文）。
- 如果该字段为 `BooleanField` 或 `NullBooleanField` ，Django 会显示漂亮的"on"或"off"图标而不是 `True` 或 `False` 。
- 如果给出的字符串是模型、 `ModelAdmin` 的一个方法或可调用对象，Django 将默认转义HTML输出。如果你不希望转义方法的输出，可以给方法一个 `allow_tags` 属性，其值为 `True` 。然而，为了避免XSS 漏洞，应该使用 `format_html()` 转义用户提供的输入。

  下面是一个完整的示例模型：

```
from django.db import models
from django.contrib import admin
from django.utils.html import format_html

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_name(self):
        return format_html('&lt;span style="color: #{};"&gt;
{} {}&lt;/span&gt;',
                           self.color_code,
                           self.first_name,
                           self.last_name)

    colored_name.allow_tags = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'colored_name
')
```

- 如果给出的字符串是模型、 `ModelAdmin` 的一个方法或一个返回 True 或 False 的可调用的方法，然后赋值给方法的 `boolean` 属性一个 `True` 值，Django 将显示漂亮的"on"或"off"图标，。

  下面是一个完整的示例模型：

```
from django.db import models
from django.contrib import admin

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    birthday = models.DateField()

    def born_in_fifties(self):
        return self.birthday.strftime('%Y')[:3] == '195'
    born_in_fifties.boolean = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'born_in_fifties')
```

- `__str__()` （Python 2 上是 `__unicode__()` ）方法在 `list_display` 中同样合法，就和任何其他模型方法一样，所以下面这样写完全OK：

```
list_display = ('__str__', 'some_other_field')
```

- 通常情况下， `list_display` 的元素如果不是实际的数据库字段不能用于排序（因为 Django 所有的排序都在数据库级别）。

  然而，如果 `list_display` 元素表示数据库的一个特定字段，你可以通过设置 元素的 `admin_order_field` 属性表示这一事实。

  例如：

```
from django.db import models
from django.contrib import admin
from django.utils.html import format_html

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_first_name(self):
        return format_html('&lt;span style="color: #{};"&gt;
{}&lt;/span&gt;',
                           self.color_code,
                           self.first_name)

    colored_first_name.allow_tags = True
    colored_first_name.admin_order_field = 'first_name'

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'colored_first_name')
```

上面的示例告诉Django 在Admin 中按照按 `colored_first_name` 排序时依据 `first_name` 字段。

New in Django 1.7.

要表示按照 `admin_order_field` 降序排序，你可以在该字段名称前面使用一个连字符前缀。使用上面的示例，这会看起来像：

```
colored_first_name.admin_order_field = '-first_name'
```

- `list_display` 的元素也可以是属性。不过请注意，由于方式属性在Python 中的工作方式，在属性上设置 `short_description` 只能使用 `property()` 函数，不 能使用 `@property` 装饰器。

  例如：

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def my_property(self):
        return self.first_name + ' ' + self.last_name
    my_property.short_description = "Full name of the person
"

    full_name = property(my_property)

class PersonAdmin(admin.ModelAdmin):
    list_display = ('full_name',)
```

- `list_display` 中的字段名称还将作为HTML 输出的CSS 类， 形式为每个 `<th>` 元素上具有 `column-<field_name>` 。例如这可以用于在CSS 文件中设置列的宽度。
- Django 会尝试以下面的顺序解释 `list_display` 的每个元素：
  - 模型的字段。
  - 可调用对象。
  - 表示 `ModelAdmin` 属性的字符串。
  - 表示模型属性的字符串。

  例如，如果 `first_name` 既是模型的一个字段又是 `ModelAdmin` 的一个属性，使用的将是模型字段。

`ModelAdmin.``list_display_links`

使用 `list_display_links` 可以控制 `list_display` 中的字段是否应该链接到对象的“更改”页面。

默认情况下，更改列表页将链接第一列 - `list_display` 中指定的第一个字段 - 到每个项目的更改页面。但是 `list_display_links` 可让您更改此设置：

- 将其设置为 `None` ，根本不会获得任何链接。
- 将其设置为要将其列转换为链接的字段列表或元组（格式与 `list_display` 相同）。

  您可以指定一个或多个字段。只要这些字段出现在 `list_display` 中，Django不会关心多少（或多少）字段被链接。唯一的要求是，如果要以这种方式使用 `list_display_links` ，则必须定义 `list_display` 。

在此示例中， `first_name` 和 `last_name` 字段将链接到更改列表页面上：

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'birthday')
    list_display_links = ('first_name', 'last_name')
```

在此示例中，更改列表页面网格将没有链接：

```
class AuditEntryAdmin(admin.ModelAdmin):
    list_display = ('timestamp', 'message')
    list_display_links = None
```

Changed in Django 1.7:

`None` 作为有效的 `list_display_links` 值添加。

`ModelAdmin.``list_editable`

将 `list_editable` 设置为模型上的字段名称列表，这将允许在更改列表页面上进行编辑。也就是说， `list_editable` 中列出的字段将在更改列表页面上显示为表单小部件，允许用户一次编辑和保存多行。

注意

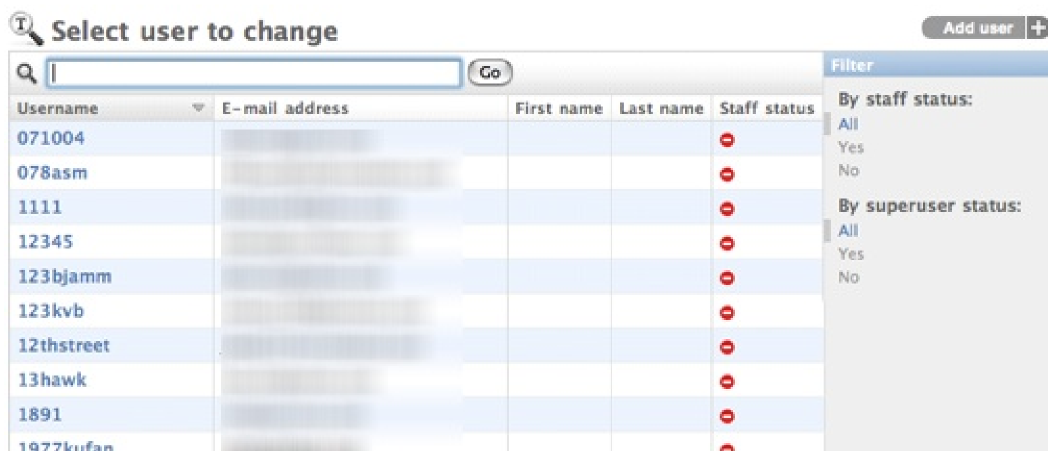
`list_editable` 以特定方式与其他几个选项进行交互；您应该注意以下规则：

- `list_editable` 中的任何字段也必须位于 `list_display` 中。您无法编辑未显示的字段！
- 同一字段不能在 `list_editable` 和 `list_display_links` 中列出 - 字段不能同时是表单和链接。

如果这些规则中的任一个损坏，您将收到验证错误。

`ModelAdmin.``list_filter`

`list_filter` 设置激活激活Admin 修改列表页面右侧栏中的过滤器，如下面的屏幕快照所示：



`list_filter` 应该是一个列表或元组，其每个元素应该是下面类型中的一种：

- 字段名称，其指定的字段应该是 `BooleanField` 、 `CharField` 、 `DateField` 、 `DateTimeField` 、 `Inte` 、 `ForeignKey` 或 `ManyToManyField` ，例如：

```
class PersonAdmin(admin.ModelAdmin):
    list_filter = ('is_staff', 'company')
```

  `list_filter` 中的字段名称也可以使用 `__` 查找跨关联关系，例如：

```
class PersonAdmin(admin.UserAdmin):
    list_filter = ('company__name',)
```

- 一个继承自 `django.contrib.admin.SimpleListFilter` 的类，你需要给它提供 `title` 和 `parameter_name` 属性来重写 `lookups` 和 `queryset` 方法，例如：

```
from datetime import date

from django.contrib import admin
from django.utils.translation import ugettext_lazy as _

class DecadeBornListFilter(admin.SimpleListFilter):
    # Human-readable title which will be displayed in the
    # right admin sidebar just above the filter options.
    title = _('decade born')

    # Parameter for the filter that will be used in the URL
query.
    parameter_name = 'decade'

    def lookups(self, request, model_admin):
        """
 Returns a list of tuples. The first element in each
 tuple is the coded value for the option that will
 appear in the URL query. The second element is the
 human-readable name for the option that will appear
 in the right sidebar.
 """
        return (
            ('80s', _('in the eighties')),
            ('90s', _('in the nineties')),
        )

    def queryset(self, request, queryset):
        """
 Returns the filtered queryset based on the value
 provided in the query string and retrievable via
 `self.value()`.
 """
        # Compare the requested value (either '80s' or '90s'
)
        # to decide how to filter the queryset.
        if self.value() == '80s':
            return queryset.filter(birthday__gte=date(1980,
1, 1),
                                    birthday__lte=date(1989,
 12, 31))
        if self.value() == '90s':
            return queryset.filter(birthday__gte=date(1990,
1, 1),
                                    birthday__lte=date(1999,
 12, 31))

class PersonAdmin(admin.ModelAdmin):
    list_filter = (DecadeBornListFilter,)
```

注

作为一种方便，`HttpRequest` 对象将传递给 `lookups` 和 `queryset` 方法，例如：

```
class AuthDecadeBornListFilter(DecadeBornListFilter):

    def lookups(self, request, model_admin):
        if request.user.is_superuser:
            return super(AuthDecadeBornListFilter,
                self).lookups(request, model_admin)

    def queryset(self, request, queryset):
        if request.user.is_superuser:
            return super(AuthDecadeBornListFilter,
                self).queryset(request, queryset)
```

也作为一种方便，`ModelAdmin` 对象将传递给 `lookups` 方法，例如如果你想要基于现有的数据查找：

```
class AdvancedDecadeBornListFilter(DecadeBornListFilter):

    def lookups(self, request, model_admin):
        """
 Only show the lookups if there actually is
 anyone born in the corresponding decades.
 """
        qs = model_admin.get_queryset(request)
        if qs.filter(birthday__gte=date(1980, 1, 1),
                      birthday__lte=date(1989, 12, 31)).exis
ts():
            yield ('80s', _('in the eighties'))
        if qs.filter(birthday__gte=date(1990, 1, 1),
                      birthday__lte=date(1999, 12, 31)).exis
ts():
            yield ('90s', _('in the nineties'))
```

- 一个元组，第一个元素是字段名称，第二个元素是从继承自 `django.contrib.admin.FieldListFilter` 的一个类，例如：

```
class PersonAdmin(admin.ModelAdmin):
    list_filter = (
        ('is_staff', admin.BooleanFieldListFilter),
    )
```

New in Django 1.8.

你可以使用 `RelatedOnlyFieldListFilter` 限制与该对象关联的模型的选项：

```
class BookAdmin(admin.ModelAdmin):
    list_filter = (
        ('author', admin.RelatedOnlyFieldListFilter),
    )
```

假设 `author` 是 `User` 模型的一个 `ForeignKey` ，这将限制 `list_filter` 的选项为编写过书籍的用户，而不是所有用户。

注意

`FieldListFilter` API 被视为内部的，可能会改变。

也可以指定自定义模板用于渲染列表筛选器：

```
class FilterWithCustomTemplate(admin.SimpleListFilter):
    template = "custom_template.html"
```

具体的例子请参见Django 提供的默认模板（ `admin/filter.html` ）。

`ModelAdmin.``list_max_show_all`

设置 `list_max_show_all` 以控制在“显示所有”管理更改列表页面上可以显示的项目数。只有当总结果计数小于或等于此设置时，管理员才会在更改列表上显示“显示全部”链接。默认情况下，设置为 `200` 。

`ModelAdmin.``list_per_page`

`list_per_page` 设置控制Admin 修改列表页面每页中显示多少项。默认设置为 `100` 。

`ModelAdmin.``list_select_related`

设置 `list_select_related` 以告诉Django在检索管理更改列表页面上的对象列表时使用 `select_related()` 。这可以节省大量的数据库查询。

该值应该是布尔值，列表或元组。默认值为 `False` 。

当值为 `True` 时，将始终调用 `select_related()` 。When value is set to `False` , Django will look at `list_display` and call `select_related()` if any `ForeignKey` is present.

如果您需要更细粒度的控制，请使用元组（或列表）作为 `list_select_related` 的值。空元组将阻止Django调用 `select_related` 。任何其他元组将直接传递到 `select_related` 作为参数。例如：

```
class ArticleAdmin(admin.ModelAdmin):
    list_select_related = ('author', 'category')
```

将会调用 `select_related('author', 'category')` .

`ModelAdmin.``ordering`

设置 `ordering` 以指定如何在Django管理视图中对对象列表进行排序。这应该是与模型的 `ordering` 参数格式相同的列表或元组。

如果没有提供，Django管理员将使用模型的默认排序。

如果您需要指定动态顺序（例如，根据用户或语言），您可以实施 `get_ordering()` 方法。

`ModelAdmin.``paginator`

paginator类用于分页。默认情况下，使用 `django.core.paginator.Paginator` 。如果自定义paginator类没有与 `django.core.paginator.Paginator` 相同的构造函数接口，则还需要为 `ModelAdmin.get_paginator()` 。

`ModelAdmin.``prepopulated_fields`

将 `prepopulated_fields` 设置为将字段名称映射到其应预先填充的字段的字典：

```
class ArticleAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

设置时，给定字段将使用一些JavaScript来从分配的字段填充。此功能的主要用途是自动从一个或多个其他字段生成 `SlugField` 字段的值。生成的值是通过连接源字段的值，然后将该结果转换为有效的字节（例如用空格替换破折号）来生成的。

`prepopulated_fields` 不接受 `DateTimeField` ， `ForeignKey` 或 `ManyToManyField` 字段。

`ModelAdmin.``preserve_filters`

管理员现在在创建，编辑或删除对象后保留列表视图中的过滤器。您可以将此属性设置为 `False` ，以恢复之前清除过滤器的行为。

`ModelAdmin.``radio_fields`

By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey` or have `choices` set. 如果 `radio_fields` 中存在字段，Django将使用单选按钮接口。假设 `group` 是 `Person` 模型上的 `ForeignKey`

```
class PersonAdmin(admin.ModelAdmin):
    radio_fields = {"group": admin.VERTICAL}
```

您可以选择使用 `django.contrib.admin` 模块中的 `HORIZONTAL` 或 `VERTICAL` 。

除非是 `ForeignKey` 或设置了 `choices` ，否则不要在 `radio_fields` 中包含字段。

`ModelAdmin.``raw_id_fields`

默认情况下，Django 的Admin 对 `ForeignKey` 字段使用选择框表示 (<select>) 。有时候你不想在下拉菜单中显示所有相关实例产生的开销。

`raw_id_fields` 是一个字段列表，你希望将 `ForeignKey` 或 `ManyToManyField` 转换成 `Input` Widget：

```
class ArticleAdmin(admin.ModelAdmin):
    raw_id_fields = ("newspaper",)
```

如果该字段是一个 `ForeignKey` ， `raw_id_fields` `Input` Widget 应该包含一个外键，或者如果字段是一个 `ManyToManyField` 则应该是一个逗号分隔的值的列表。 `raw_id_fields` Widget 在字段旁边显示一个放大镜按钮，允许用户搜索并选择一个值：

Groups: 1,2

`ModelAdmin.``readonly_fields`

默认情况下，管理员将所有字段显示为可编辑。此选项中的任何字段（应为 `list` 或 `tuple` ）将按原样显示其数据，且不可编辑；它们也会从用于创建和编辑的 `ModelForm` 中排除。请注意，指定 `ModelAdmin.fields` 或 `ModelAdmin.fieldsets` 时，只读字段必须显示才能显示（否则将被忽略）。

如果在未通过 `ModelAdmin.fields` 或 `ModelAdmin.fieldsets` 定义显式排序的情况下使用 `readonly_fields` ，则它们将在所有可编辑字段之后添加。

只读字段不仅可以显示模型字段中的数据，还可以显示模型方法的输出或 `ModelAdmin` 类本身的方法。这与 `ModelAdmin.list_display` 的行为非常相似。这提供了一种使用管理界面提供对正在编辑的对象的状态的反馈的简单方法，例如：

```
from django.contrib import admin
from django.utils.html import format_html_join
from django.utils.safestring import mark_safe

class PersonAdmin(admin.ModelAdmin):
    readonly_fields = ('address_report',)

    def address_report(self, instance):
        # assuming get_full_address() returns a list of strings
        # for each line of the address and you want to separate
each
        # line by a linebreak
        return format_html_join(
            mark_safe('&lt;br/&gt;'),
            '{}',
            ((line,) for line in instance.get_full_address()),
        ) or "&lt;span class='errors'&gt;I can't determine this
address.&lt;/span&gt;"

    # short_description functions like a model field's verbose_n
ame
    address_report.short_description = "Address"
    # in this example, we have used HTML tags in the output
    address_report.allow_tags = True
```

`ModelAdmin.``save_as`

`save_as` 设置启用Admin 更改表单上的“save as”功能。

通常情况下，对象有三个保存选项："保存"、"保存并继续编辑"和"保存并添加另一个"。如果 `save_as` 为 `True` ，"保存并添加另一个"将由"另存为"按钮取代。

"另存为"表示对象将被保存为一个新的对象 （带有一个新的 ID)，而不是旧的对象。

默认情况下， `save_as` 设置为 `False` 。

`ModelAdmin.``save_on_top`

设置 `save_on_top` 可在表单顶部添加保存按钮。

通常，保存按钮仅出现在表单的底部。如果您设置 `save_on_top` ，则按钮将同时显示在顶部和底部。

默认情况下， `save_on_top` 设置为 `False` 。

`ModelAdmin.``search_fields`

`search_fields` 设置启用Admin 更改列表页面上的搜索框。此属性应设置为每当有人在该文本框中提交搜索查询将搜索的字段名称的列表。

这些字段应该是某种文本字段，如 `CharField` 或 `TextField` 。你还可以通过查询API 的"跟随"符号进行 `ForeignKey` 或 `ManyToManyField` 上的关联查找：

```
search_fields = ['foreign_key__related_fieldname']
```

例如，如果你有一个具有作者的博客，下面的定义将启用通过作者的电子邮件地址搜索博客条目：

```
search_fields = ['user__email']
```

如果有人在Admin 搜索框中进行搜索，Django 拆分搜索查询为单词并返回包含每个单词的所有对象，不区分大小写，其中每个单词必须在至少一个 `search_fields` 。例如，如果 `search_fields` 设置为 `['first_name', 'last_name']` ，用户搜索 `john lennon` ，Django 的行为将相当于下面的这个 `WHERE` SQL 子句：

```
WHERE (first_name ILIKE '%john%' OR last_name ILIKE '%john%')
AND (first_name ILIKE '%lennon%' OR last_name ILIKE '%lennon%')
```

若要更快和/或更严格的搜索，请在字典名称前面加上前缀：

`^`

匹配字段的开始。例如，如果 `search_fields` 设置为 `['^first_name', '^last_name']` ，用户搜索 `john lennon` 时，Django 的行为将等同于下面这个 `WHERE` SQL 字句：

```
WHERE (first_name ILIKE 'john%' OR last_name ILIKE 'john%')
AND (first_name ILIKE 'lennon%' OR last_name ILIKE 'lennon%')
```

此查询比正常 `'%john%'` 查询效率高，因为数据库只需要检查某一列数据的开始，而不用寻找整列数据。另外，如果列上有索引，有些数据库可能能够对于此查询使用索引，即使它是 `like` 查询。

`=`

精确匹配，不区分大小写。例如，如果 `search_fields` 设置为 `['=first_name', '=last_name']` ，用户搜索 `john lennon` 时，Django 的行为将等同于下面这个 `WHERE` SQL 字句：

```
WHERE (first_name ILIKE 'john' OR last_name ILIKE 'john')
AND (first_name ILIKE 'lennon' OR last_name ILIKE 'lennon')
```

注意，该查询输入通过空格分隔，所以根据这个示例，目前不能够搜索 `first_name` 精确匹配 `'john winston'`（包含空格）的所有记录。

`@`

Performs a full-text match. This is like the default search method but uses an index. Currently this is only available for MySQL.

如果你需要自定义搜索，你可以使用 `ModelAdmin.get_search_results()` 来提供附件的或另外一种搜索行为。

`ModelAdmin.``show_full_result_count`

New in Django 1.8.

设置 `show_full_result_count` 以控制是否应在过滤的管理页面上显示对象的完整计数（例如 `99 结果 103 total）`）。如果此选项设置为 `False`，则像 `99 结果 （显示 ）`。

默认情况下，`show_full_result_count=True` 生成一个查询，对表执行完全计数，如果表包含大量行，这可能很昂贵。

`ModelAdmin.``view_on_site`

New in Django 1.7.

设置 `view_on_site` 以控制是否显示“在网站上查看”链接。此链接将带您到一个URL，您可以在其中显示已保存的对象。

此值可以是布尔标志或可调用的。如果 `True`（默认值），对象的 `get_absolute_url()` 方法将用于生成网址。

如果您的模型有 `get_absolute_url()` 方法，但您不想显示“在网站上查看”按钮，则只需将 `view_on_site` 设置为 `False`：

```
from django.contrib import admin

class PersonAdmin(admin.ModelAdmin):
    view_on_site = False
```

如果它是可调用的，它接受模型实例作为参数。例如：

```
from django.contrib import admin
from django.core.urlresolvers import reverse

class PersonAdmin(admin.ModelAdmin):
    def view_on_site(self, obj):
        return 'http://example.com' + reverse('person-detail',
                                              kwargs={'slug': ob
j.slug})
```

自定义模板的选项

重写Admin模板 一节描述如何重写或扩展默认Admin 模板。使用以下选项来重写 `ModelAdmin` 视图使用的默认模板：

`ModelAdmin.``add_form_template`

`add_view()` 使用的自定义模板的路径。

`ModelAdmin.``change_form_template`

`change_view()` 使用的自定义模板的路径。

`ModelAdmin.``change_list_template`

`changelist_view()` 使用的自定义模板的路径。

`ModelAdmin.``delete_confirmation_template`

`delete_view()` 使用的自定义模板，用于删除一个或多个对象时显示一个确认页。

`ModelAdmin.``delete_selected_confirmation_template`

`delete_selected()` 使用的自定义模板，用于删除一个或多个对象时显示一个确认页。参见*Action* 的文档。

`ModelAdmin.``object_history_template`

`history_view()` 使用的自定义模板的路径。

ModelAdmin methods

警告

`ModelAdmin.save_model()` 以及 `ModelAdmin.delete_model()` must save/delete the object, they are not for veto purposes, rather they allow you to perform extra operations.

`ModelAdmin.``save_model` (*request*, *obj*, *form*, *change*)

`save_model` 方法被赋予 `HttpRequest` ，模型实例， `ModelForm` 实例和布尔值，基于它是添加还是更改对象。在这里您可以执行任何预保存或后保存操作。

例如，在保存之前将 `request.user` 附加到对象：

```
from django.contrib import admin

class ArticleAdmin(admin.ModelAdmin):
    def save_model(self, request, obj, form, change):
        obj.user = request.user
        obj.save()
```

`ModelAdmin.``delete_model` (*request*, *obj*)

`delete_model` 方法给出了 `HttpRequest` 和模型实例。使用此方法执行预删除或后删除操作。

`ModelAdmin.``save_formset` (*request*, *form*, *formset*, *change*)

`save_formset` 方法是给予 `HttpRequest` ，父 `ModelForm` 实例和基于是否添加或更改父对象的布尔值。

例如，要将 `request.user` 附加到每个已更改的formset模型实例：

```
class ArticleAdmin(admin.ModelAdmin):
    def save_formset(self, request, form, formset, change):
        instances = formset.save(commit=False)
        for obj in formset.deleted_objects:
            obj.delete()
        for instance in instances:
            instance.user = request.user
            instance.save()
        formset.save_m2m()
```

另请参见*Saving objects in the formset*。

`ModelAdmin.``get_ordering` (*request*)

The `get_ordering` method takes a `request` as parameter and is expected to return a `list` or `tuple` for ordering similar to the `ordering` attribute. 例如：

```
class PersonAdmin(admin.ModelAdmin):

    def get_ordering(self, request):
        if request.user.is_superuser:
            return ['name', 'rank']
        else:
            return ['name']
```

`ModelAdmin.``get_search_results` (*request*, *queryset*, *search_term*)

`get_search_results` 方法将显示的对象列表修改为与提供的搜索项匹配的对象。它接受请求，应用当前过滤器的查询集以及用户提供的搜索项。它返回一个包含被修改以实现搜索的查询集的元组，以及一个指示结果是否可能包含重复项的布尔值。

默认实现搜索在 `ModelAdmin.search_fields` 中命名的字段。

此方法可以用您自己的自定义搜索方法覆盖。例如，您可能希望通过整数字段搜索，或使用外部工具（如Solr或Haystack）。您必须确定通过搜索方法实现的查询集更改是否可能在结果中引入重复项，并在返回值的第二个元素中返回 `True` 。

例如，要启用按整数字段搜索，您可以使用：

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'age')
    search_fields = ('name',)

    def get_search_results(self, request, queryset, search_term)
:
        queryset, use_distinct = super(PersonAdmin, self).get_se
arch_results(request, queryset, search_term)
        try:
            search_term_as_int = int(search_term)
        except ValueError:
            pass
        else:
            queryset |= self.model.objects.filter(age=search_ter
m_as_int)
        return queryset, use_distinct
```

`ModelAdmin.``save_related` (*request*, *form*, *formsets*, *change*)

`save_related` 方法给出了 `HttpRequest` ，父 `ModelForm` 实例，内联表单列表和一个布尔值，添加或更改。在这里，您可以对与父级相关的对象执行任何预保存或后保存操作。请注意，此时父对象及其形式已保存。

`ModelAdmin.``get_readonly_fields` (*request*, *obj=None*)

`get_readonly_fields` 方法在添加表单上给予 `HttpRequest` 和 `obj` （或 `None` ），希望返回将以只读形式显示的字段名称的 `list` 或 `tuple` ，如上面在 `ModelAdmin.readonly_fields` 部分中所述。

`ModelAdmin.``get_prepopulated_fields` (*request*, *obj=None*)

`get_prepopulated_fields` 方法在添加表单上给予 `HttpRequest` 和 `obj` （或 `None` ），预期返回 `dictionary` ，如上面在 `ModelAdmin.prepopulated_fields` 部分中所述。

`ModelAdmin.``get_list_display` (*request*)

`get_list_display` 方法被赋予 `HttpRequest` ，并且希望返回字段名称的 `list` 或 `tuple` 显示在如上所述的 `ModelAdmin.list_display` 部分中的changelist视图上。

`ModelAdmin.``get_list_display_links` (*request*, *list_display*)

The get_list_display_links method is given the HttpRequest and the list or tuple returned by ModelAdmin.get_list_display() . 预期将返回更改列表上将链接到更改视图的字段名称的 None 或 list 或 tuple ，如上所述在 ModelAdmin.list_display_links 部分中。

Changed in Django 1.7:

None 作为有效的 get_list_display_links() 返回值添加。

ModelAdmin.``get_fields (*request*, *obj=None*)

New in Django 1.7.

get_fields 方法被赋予 HttpRequest 和 obj 被编辑（或在添加表单上 None ），希望返回字段列表，如上面在 ModelAdmin.fields 部分中所述。

ModelAdmin.``get_fieldsets (*request*, *obj=None*)

get_fieldsets 方法是在添加表单上给予 HttpRequest 和 obj （或 None ），期望返回二元组列表，其中每个二元组在管理表单页面上表示 <fieldset> ，如上面在 ModelAdmin.fieldsets 部分。

ModelAdmin.``get_list_filter (*request*)

get_list_filter 方法被赋予 HttpRequest ，并且期望返回与 list_filter 属性相同类型的序列类型。

ModelAdmin.``get_search_fields (*request*)

New in Django 1.7.

get_search_fields 方法被赋予 HttpRequest ，并且期望返回与 search_fields 属性相同类型的序列类型。

ModelAdmin.``get_inline_instances (*request*, *obj=None*)

get_inline_instances 方法在添加表单上给予 HttpRequest 和 obj （或 None ），预期会返回 list 或 tuple 的 InlineModelAdmin 对象，如下面的 InlineModelAdmin 部分所述。例如，以下内容将返回内联，而不进行基于添加，更改和删除权限的默认过滤：

```
class MyModelAdmin(admin.ModelAdmin):
    inlines = (MyInline,)

    def get_inline_instances(self, request, obj=None):
        return [inline(self.model, self.admin_site) for inline i
n self.inlines]
```

如果覆盖此方法，请确保返回的内联是 `inlines` 中定义的类的实例，或者在添加相关对象时可能会遇到“错误请求”错误。

`ModelAdmin.``get_urls` ()

`ModelAdmin` 的 `get_urls` 方法返回ModelAdmin 将要用到的URLs，方式与URLconf 相同。因此，你可以用*URL 调度器* 中所述的方式扩展它们：

```
class MyModelAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super(MyModelAdmin, self).get_urls()
        my_urls = [
            url(r'^my_view/$', self.my_view),
        ]
        return my_urls + urls

    def my_view(self, request):
        # ...
        context = dict(
           # Include common variables for rendering the admin te
mplate.
           self.admin_site.each_context(request),
           # Anything else you want in the context...
           key=value,
        )
        return TemplateResponse(request, "sometemplate.html", co
ntext)
```

如果你想要使用Admin 的布局，可以从 `admin/base_site.html` 扩展：

```
{% extends "admin/base_site.html" %}
{% block content %}
...
{% endblock %}
```

注

请注意，自定义的模式包含在正常的Admin URLs之前：Admin URL 模式非常宽松，将匹配几乎任何内容，因此你通常要追加自定义的URLs 到内置的URLs 前面。

在此示例中， `my_view` 的访问点将是 `/admin/myapp/mymodel/my_view/` （假设Admin URLs 包含在 `/admin/` 下）。

但是, 上述定义的函数 `self.my_view` 将遇到两个问题：

- 它不 执行任何权限检查，所以会向一般公众开放。
- 它不提供任何HTTP头的详细信息以防止缓存。这意味着，如果页面从数据库检索数据，而且缓存中间件处于活动状态，页面可能显示过时的信息。

因为这通常不是你想要的，Django 提供一个方便的封装函数来检查权限并标记视图为不可缓存的。这个封装函数就是 `AdminSite.admin_view()` （例如位于 `ModelAdmin` 实例中的 `self.admin_site.admin_view` ）；就像这样使用它：

```
class MyModelAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super(MyModelAdmin, self).get_urls()
        my_urls = [
            url(r'^my_view/$', self.admin_site.admin_view(self.m
y_view))
        ]
        return my_urls + urls
```

请注意上述第5行中的被封装的视图：

```
url(r'^my_view/$', self.admin_site.admin_view(self.my_view))
```

这个封装将保护 `self.my_view` 免受未经授权的访问，并将运用 `django.views.decorators.cache.never_cache` 装饰器以确保它不会被缓存，即使缓存中间件是活跃的。

如果该页面是可缓存的，但你仍然想要执行权限检查，你可以传递 `AdminSite.admin_view()` 的 `cacheable=True` 参数：

```
url(r'^my_view/$', self.admin_site.admin_view(self.my_view, cach
eable=True))
```

`ModelAdmin.``get_form` (*request*, *obj=None*, ***kwargs*)

返回Admin中添加和更改视图使用的 `ModelForm` 类，请参阅 `add_view()` 和 `change_view()` 。

其基本的实现是使用 `modelform_factory()` 来子类化 `form` ，修改如 `fields` 和 `exclude` 属性。所以，举个例子，如果你想要为超级用户提供额外的字段，你可以换成不同的基类表单，就像这样：

```
class MyModelAdmin(admin.ModelAdmin):
    def get_form(self, request, obj=None, **kwargs):
        if request.user.is_superuser:
            kwargs['form'] = MySuperuserForm
        return super(MyModelAdmin, self).get_form(request, obj,
**kwargs)
```

你也可以简单地直接返回一个自定义的 `ModelForm` 类。

`ModelAdin.``get_formsets` (*request*, *obj=None*)

自1.7版起已弃用：请改用 `get_formsets_with_inlines()` 。

产生 `InlineModelAdmin` 用于管理员添加和更改视图。

例如，如果您只想在更改视图中显示特定的内联，则可以覆盖 `get_formsets` ，如下所示：

```
class MyModelAdmin(admin.ModelAdmin):
    inlines = [MyInline, SomeOtherInline]

    def get_formsets(self, request, obj=None):
        for inline in self.get_inline_instances(request, obj):
            # hide MyInline in the add view
            if isinstance(inline, MyInline) and obj is None:
                continue
            yield inline.get_formset(request, obj)
```

`ModelAdmin.``get_formsets_with_inlines` (*request*, *obj=None*)

New in Django 1.7.

产量（ `FormSet` ， `InlineModelAdmin` ）对用于管理添加和更改视图。

例如，如果您只想在更改视图中显示特定的内联，则可以覆盖 `get_formsets_with_inlines` ，如下所示：

```
class MyModelAdmin(admin.ModelAdmin):
    inlines = [MyInline, SomeOtherInline]

    def get_formsets_with_inlines(self, request, obj=None):
        for inline in self.get_inline_instances(request, obj):
            # hide MyInline in the add view
            if isinstance(inline, MyInline) and obj is None:
                continue
            yield inline.get_formset(request, obj), inline
```

`ModelAdmin.``formfield_for_foreignkey` (*db_field*, *request*, ***kwargs*)

`ModelAdmin` 上的 `formfield_for_foreignkey` 方法允许覆盖外键字段的默认窗体字段。例如，要根据用户返回此外键字段的对象子集：

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_foreignkey(self, db_field, request, **kwar
gs):
        if db_field.name == "car":
            kwargs["queryset"] = Car.objects.filter(owner=reques
t.user)
        return super(MyModelAdmin, self).formfield_for_foreignke
y(db_field, request, **kwargs)
```

这使用 `HttpRequest` 实例过滤 `Car` 外键字段，只显示由 `User` 实例拥有的汽车。

`ModelAdmin.``formfield_for_manytomany` (*db_field*, *request*, ***kwargs*)

与 `formfield_for_foreignkey` 方法类似，可以覆盖 `formfield_for_manytomany` 方法来更改多对多字段的默认窗体字段。例如，如果所有者可以拥有多个汽车，并且汽车可以属于多个所有者 - 多对多关系，则您可以过滤 `Car` 外键字段，仅显示由 `User` ：

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_manytomany(self, db_field, request, **kwar
gs):
        if db_field.name == "cars":
            kwargs["queryset"] = Car.objects.filter(owner=reques
t.user)
        return super(MyModelAdmin, self).formfield_for_manytoman
y(db_field, request, **kwargs)
```

`ModelAdmin.``formfield_for_choice_field` (*db_field*, *request*, ***kwargs*)

与 `formfield_for_foreignkey` 和 `formfield_for_manytomany` 方法类似，可以覆盖 `formfield_for_choice_field` 方法更改已声明选择的字段的默认窗体字段。例如，如果超级用户可用的选择应与正式工作人员可用的选项不同，则可按以下步骤操作：

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_choice_field(self, db_field, request, **kw
args):
        if db_field.name == "status":
            kwargs['choices'] = (
                ('accepted', 'Accepted'),
                ('denied', 'Denied'),
            )
            if request.user.is_superuser:
                kwargs['choices'] += (('ready', 'Ready for deplo
yment'),)
        return super(MyModelAdmin, self).formfield_for_choice_fi
eld(db_field, request, **kwargs)
```

注意

在表单字段上设置的任何 `choices` 属性将仅限于表单字段。如果模型上的相应字段有选择集，则提供给表单的选项必须是这些选择的有效子集，否则，在保存模型本身之前验证模型本身时，表单提交将失败并显示 `ValidationError` 。

`ModelAdmin.``get_changelist` (*request*, ***kwargs*)

返回要用于列表的 `Changelist` 类。默认情况下，使用 `django.contrib.admin.views.main.ChangeList` 。通过继承此类，您可以更改列表的行为。

`ModelAdmin.``get_changelist_form` (*request*, ***kwargs*)

返回 `ModelForm` 类以用于更改列表页面上的 `Formset` 。要使用自定义窗体，例如：

```
from django import forms

class MyForm(forms.ModelForm):
    pass

class MyModelAdmin(admin.ModelAdmin):
    def get_changelist_form(self, request, **kwargs):
        return MyForm
```

注意

如果您在 `ModelForm` 上定义 `Meta.model` 属性，则还必须定义 `Meta.fields` 属性（或 `Meta.exclude` 属性）。但是， `ModelAdmin` 会忽略此值，并使用 `ModelAdmin.list_editable` 属性覆盖该值。最简单的解决方案是省略 `Meta.model` 属性，因为 `ModelAdmin` 将提供要使用的正确模型。

`ModelAdmin.``get_changelist_formset` (*request*, ***kwargs*)

如果使用 `list_editable` ，则返回*ModelFormSet*类以在更改列表页上使用。要使用自定义表单集，例如：

```
from django.forms.models import BaseModelFormSet

class MyAdminFormSet(BaseModelFormSet):
    pass

class MyModelAdmin(admin.ModelAdmin):
    def get_changelist_formset(self, request, **kwargs):
        kwargs['formset'] = MyAdminFormSet
        return super(MyModelAdmin, self).get_changelist_formset(
request, **kwargs)
```

`ModelAdmin.``has_add_permission` (*request*)

如果允许添加对象，则应返回 `True` ，否则返回 `False` 。

`ModelAdmin.``has_change_permission` (*request*, *obj=None*)

如果允许编辑obj，则应返回 `True` ，否则返回 `False` 。如果obj为 `None` ，则应返回 `True` 或 `False` 以指示是否允许对此类对象进行编辑（例如， `False` 将被解释为意味着当前用户不允许编辑此类型的任何对象）。

`ModelAdmin.``has_delete_permission` (*request*, *obj=None*)

如果允许删除obj，则应返回 `True` ，否则返回 `False` 。If obj is `None` , should return `True` or `False` to indicate whether deleting objects of this type is permitted in general (e.g., `False` will be interpreted as meaning that the current user is not permitted to delete any object of this type).

`ModelAdmin.``has_module_permission` (*request*)

New in Django 1.8.

如果在管理索引页上显示模块并允许访问模块的索引页，则应返回 `True` ，否则 `False` 。默认情况下使用 `User.has_module_perms()` 。覆盖它不会限制对添加，更改或删除视图的访问， `has_add_permission()` ， `has_change_permission()` 和 `has_delete_pe`
用于那。

`ModelAdmin.``get_queryset` (*request*)

`ModelAdmin` 上的 `get_queryset` 方法会返回管理网站可以编辑的所有模型实例的 `QuerySet` 。覆盖此方法的一个用例是显示由登录用户拥有的对象：

```
class MyModelAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        qs = super(MyModelAdmin, self).get_queryset(request)
        if request.user.is_superuser:
            return qs
        return qs.filter(author=request.user)
```

`ModelAdmin.``message_user` (*request*, *message*, *level=messages.INFO*, *extra_tags=''*, *fail_silently=False*)

使用 `django.contrib.messages` 向用户发送消息。参见自定义*ModelAdmin* 示例。

关键字参数运行你修改消息的级别、添加CSS 标签，如果 `contrib.messages` 框架没有安装则默默的失败。关键字参数与 `django.contrib.messages.add_message()` 的参数相匹配，更多细节请参见这个函数的文档。有一个不同点是级别除了使用整数/常数传递之外还以使用字符串。

`ModelAdmin.``get_paginator` (*queryset*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)

返回要用于此视图的分页器的实例。默认情况下，实例化 `paginator` 的实例。

`ModelAdmin.``response_add` (*request*, *obj*, *post_url_continue=None*)

为 `add_view()` 阶段确定 `HttpResponse` 。

`response_add` 在管理表单提交后，在对象和所有相关实例已创建并保存之后调用。您可以覆盖它以在对象创建后更改默认行为。

`ModelAdmin.``response_change` (*request*, *obj*)

确定 `change_view()` 阶段的 `HttpResponse` 。

`response_change` 在Admin 表单提交并保存该对象和所有相关的实例之后调用。您可以重写它来更改对象修改之后的默认行为。

`ModelAdmin.``response_delete` (*request*, *obj_display*, *obj_id*)

New in Django 1.7.

为 `delete_view()` 阶段确定 `HttpResponse` 。

在对象已删除后调用 `response_delete` 。您可以覆盖它以在对象被删除后更改默认行为。

`obj_display` 是具有已删除对象名称的字符串。

`obj_id` 是用于检索要删除的对象的序列化标识符。

New in Django 1.8:

已添加 `obj_id` 参数。

`ModelAdmin.``get_changeform_initial_data` (*request*)

New in Django 1.7.

用于管理员更改表单上的初始数据的挂钩。默认情况下，字段从 `GET` 参数给出初始值。例如， `?name=initial_value` 会将 `name` 字段的初始值设置为 `initial_value` 。

此方法应返回 `{'fieldname': 'fieldval'}` 形式的字典：

```
def get_changeform_initial_data(self, request):
    return {'name': 'custom_initial_value'}
```

其他方法

`ModelAdmin.``add_view` (*request*, *form_url=''*, *extra_context=None*)

Django视图为模型实例添加页面。见下面的注释。

`ModelAdmin.``change_view` (*request*, *object_id*, *form_url=''*, *extra_context=None*)

Django视图为模型实例版本页。见下面的注释。

`ModelAdmin.``changelist_view` (*request*, *extra_context=None*)

Django视图为模型实例更改列表/操作页面。见下面的注释。

`ModelAdmin.``delete_view` (*request*, *object_id*, *extra_context=None*)

模型实例删除确认页面的Django 视图。请参阅下面的注释。

`ModelAdmin.``history_view` (*request*, *object_id*, *extra_context=None*)

显示给定模型实例的修改历史的页面的Django视图。

与上一节中详述的钩型 `ModelAdmin` 方法不同，这五个方法实际上被设计为从管理应用程序URL调度处理程序调用为Django视图，以呈现处理模型实例的页面CRUD操作。因此，完全覆盖这些方法将显着改变管理应用程序的行为。

覆盖这些方法的一个常见原因是增加提供给呈现视图的模板的上下文数据。在以下示例中，覆盖更改视图，以便为渲染的模板提供一些额外的映射数据，否则这些数据将不可用：

```
class MyModelAdmin(admin.ModelAdmin):

    # A template for a very customized change view:
    change_form_template = 'admin/myapp/extras/openstreetmap_cha
nge_form.html'

    def get_osm_info(self):
        # ...
        pass

    def change_view(self, request, object_id, form_url='', extra
_context=None):
        extra_context = extra_context or {}
        extra_context['osm_data'] = self.get_osm_info()
        return super(MyModelAdmin, self).change_view(request, ob
ject_id,
            form_url, extra_context=extra_context)
```

这些视图返回 `TemplateResponse` 实例，允许您在渲染之前轻松自定义响应数据。有关详细信息，请参阅*TemplateResponse documentation*。

ModelAdmin asset definitions

有时候你想添加一些CSS和/或JavaScript到添加/更改视图。这可以通过在 `ModelAdmin` 上使用 `Media` 内部类来实现：

```
class ArticleAdmin(admin.ModelAdmin):
    class Media:
        css = {
            "all": ("my_styles.css",)
        }
        js = ("my_code.js",)
```

*staticfiles app*将 `STATIC_URL` （或 `MEDIA_URL` 如果 `STATIC_URL` 为 `None` 资产路径。相同的规则适用于表单上的*regular asset definitions on forms*。

jQuery

Django管理JavaScript使用jQuery库。

为了避免与用户提供的脚本或库冲突，Django的jQuery（版本1.11.2）命名为 `django.jQuery` 。如果您想在自己的管理JavaScript中使用jQuery而不包含第二个副本，则可以使用更改列表上的 `django.jQuery` 对象和添加/编辑视图。

Changed in Django 1.8:

嵌入式jQuery已经从1.9.1升级到1.11.2。

默认情况下，`ModelAdmin`类需要jQuery，因此除非有特定需要，否则不需要向您的`ModelAdmin`的媒体资源列表添加jQuery。例如，如果您需要将jQuery库放在全局命名空间中（例如使用第三方jQuery插件时）或者如果您需要更新的jQuery版本，则必须包含自己的副本。

Django提供了jQuery的未压缩和“缩小”版本，分别是`jquery.js`和`jquery.min.js`。

`ModelAdmin`和`InlineModelAdmin`具有`media`属性，可返回存储到JavaScript文件的路径的`Media`对象列表形式和/或格式。如果`DEBUG`是`True`，它将返回各种JavaScript文件的未压缩版本，包括`jquery.js`；如果没有，它将返回“minified”版本。

向管理员添加自定义验证

在管理员中添加数据的自定义验证是很容易的。自动管理界面重用`django.forms`，并且`ModelAdmin`类可以定义您自己的形式：

```
class ArticleAdmin(admin.ModelAdmin):
    form = MyArticleAdminForm
```

`MyArticleAdminForm`可以在任何位置定义，只要在需要的地方导入即可。现在，您可以在表单中为任何字段添加自己的自定义验证：

```
class MyArticleAdminForm(forms.ModelForm):
    def clean_name(self):
        # do something that validates your data
        return self.cleaned_data["name"]
```

重要的是你在这里使用`ModelForm`否则会破坏。有关详细信息，请参阅*custom validation*上的*forms*文档，更具体地说，*model form validation notes*。

InlineModelAdmin objects

class `InlineModelAdmin`

class `TabularInline`

class `StackedInline`

此管理界面能够在一个界面编辑多个Model。这些称为内联。假设你有这两个模型：

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=100)

class Book(models.Model):
   author = models.ForeignKey(Author)
   title = models.CharField(max_length=100)
```

The first step in displaying this intermediate model in the admin is to define an inline class for the Membership model:您可以通过在 `ModelAdmin.inlines` 中指定模型来为模型添加内联：

```
from django.contrib import admin

class BookInline(admin.TabularInline):
    model = Book

class AuthorAdmin(admin.ModelAdmin):
    inlines = [
        BookInline,
    ]
```

Django提供了两个 `InlineModelAdmin` 的子类如下:

- TabularInline
- StackedInline

这两者之间仅仅是在用于呈现他们的模板上有区别。

InlineModelAdmin options

`InlineModelAdmin` 与 `ModelAdmin` 具有许多相同的功能，并添加了一些自己的功能（共享功能实际上是在 `BaseModelAdmin` 超类中定义的）。共享功能包括：

- 形成
- fieldets
- 字段
- formfield_overrides
- 排除
- filter_horizontal
- filter_vertical
- 订购
- prepopulated_fields
- get_queryset()
- radio_fields
- readonly_fields

- raw_id_fields
- formfield_for_choice_field()
- formfield_for_foreignkey()
- formfield_for_manytomany()
- has_add_permission()
- has_change_permission()
- has_delete_permission()
- has_module_permission()

`InlineModelAdmin` 类添加：

`InlineModelAdmin.``model`

内联正在使用的模型。这是必需的。

`InlineModelAdmin.``fk_name`

模型上的外键的名称。在大多数情况下，这将自动处理，但如果同一父模型有多个外键，则必须显式指定 `fk_name` 。

`InlineModelAdmin.``formset`

默认为 `BaseInlineFormSet` 。使用自己的表单可以给你很多自定义的可能性。内联围绕*model formsets*构建。

`InlineModelAdmin.``form`

`form` 的值默认为 `ModelForm` 。这是在为此内联创建表单集时传递到 `inlineformset_factory()` 的内容。

警告

在为 `InlineModelAdmin` 表单编写自定义验证时，请谨慎编写依赖于父模型功能的验证。如果父模型无法验证，则可能会处于不一致状态，如*Validation on a ModelForm*中的警告中所述。

`InlineModelAdmin.``extra`

这控制除初始形式外，表单集将显示的额外表单的数量。有关详细信息，请参阅*formsets documentation*。

对于具有启用JavaScript的浏览器的用户，提供了“添加另一个”链接，以允许除了由于 `extra` 参数提供的内容之外添加任意数量的其他内联。

如果当前显示的表单数量超过 `max_num` ，或者用户未启用JavaScript，则不会显示动态链接。

`InlineModelAdmin.get_extra()` 还允许您自定义额外表单的数量。

`InlineModelAdmin.``max_num`

这控制在内联中显示的表单的最大数量。这不直接与对象的数量相关，但如果值足够小，可以。有关详细信息，请参阅*Limiting the number of editable objects*。

`InlineModelAdmin.get_max_num()` 还允许您自定义最大数量的额外表单。

`InlineModelAdmin.``min_num`

New in Django 1.7.

这控制在内联中显示的表单的最小数量。有关详细信息，请参阅 `modelformset_factory()` 。

`InlineModelAdmin.get_min_num()` 还允许您自定义显示的表单的最小数量。

`InlineModelAdmin.``raw_id_fields`

By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey` . 有时，您不希望产生必须选择要在下拉列表中显示的所有相关实例的开销。

`raw_id_fields` 是您希望更改为 `ForeignKey` 或 `ManyToManyField` 的 `Input` 窗口小部件的字段列表：

```
class BookInline(admin.TabularInline):
    model = Book
    raw_id_fields = ("pages",)
```

`InlineModelAdmin.``template`

用于在页面上呈现内联的模板。

`InlineModelAdmin.``verbose_name`

覆盖模型的内部 `Meta` 类中找到的 `verbose_name` 。

`InlineModelAdmin.``verbose_name_plural`

覆盖模型的内部 `Meta` 类中的 `verbose_name_plural` 。

`InlineModelAdmin.``can_delete`

指定是否可以在内联中删除内联对象。默认为 `True` 。

`InlineModelAdmin.``show_change_link`

New in Django 1.8.

指定是否可以在admin中更改的内联对象具有指向更改表单的链接。默认为 `False` 。

`InlineModelAdmin.``get_formset` (*request*, *obj=None*, ***kwargs*)

返回 `BaseInlineFormSet` 类，以在管理员添加/更改视图中使用。请参阅 `ModelAdmin.get_formsets_with_inlines` 的示例。

`InlineModelAdmin.``get_extra` (*request*, *obj=None*, ***kwargs*)

返回要使用的其他内联表单的数量。默认情况下，返回 `InlineModelAdmin.extra` 属性。

覆盖此方法以编程方式确定额外的内联表单的数量。例如，这可以基于模型实例（作为关键字参数 `obj` 传递）：

```
class BinaryTreeAdmin(admin.TabularInline):
    model = BinaryTree

    def get_extra(self, request, obj=None, **kwargs):
        extra = 2
        if obj:
            return extra - obj.binarytree_set.count()
        return extra
```

`InlineModelAdmin.``get_max_num` (*request*, *obj=None*, ***kwargs*)

返回要使用的额外内联表单的最大数量。默认情况下，返回 `InlineModelAdmin.max_num` 属性。

覆盖此方法以编程方式确定内联表单的最大数量。例如，这可以基于模型实例（作为关键字参数 `obj` 传递）：

```
class BinaryTreeAdmin(admin.TabularInline):
    model = BinaryTree

    def get_max_num(self, request, obj=None, **kwargs):
        max_num = 10
        if obj.parent:
            return max_num - 5
        return max_num
```

`InlineModelAdmin.``get_min_num` (*request*, *obj=None*, ***kwargs*)

New in Django 1.7.

返回要使用的内联表单的最小数量。默认情况下，返回 `InlineModelAdmin.min_num` 属性。

覆盖此方法以编程方式确定最小内联表单数。例如，这可以基于模型实例（作为关键字参数 `obj` 传递）。

使用具有两个或多个外键的模型到同一个父模型

有时可能有多个外键到同一个模型。以这个模型为例：

```
from django.db import models

class Friendship(models.Model):
    to_person = models.ForeignKey(Person, related_name="friends"
)
    from_person = models.ForeignKey(Person, related_name="from_f
riends")
```

如果您想在 `Person` 管理员添加/更改页面上显示内联，则需要明确定义外键，因为它无法自动执行：

```
from django.contrib import admin
from myapp.models import Friendship

class FriendshipInline(admin.TabularInline):
    model = Friendship
    fk_name = "to_person"

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        FriendshipInline,
    ]
```

使用多对多模型

默认情况下，多对多关系的管理窗口小部件将显示在包含 `ManyToManyField` 的实际引用的任何模型上。根据您的 `ModelAdmin` 定义，模型中的每个多对多字段将由标准HTML &lt； select multiple> t4> ，水平或垂直过滤器或 `raw_id_admin` 小部件。但是，也可以用内联替换这些小部件。

假设我们有以下模型：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, related_name='group
s')
```

如果要使用内联显示多对多关系，可以通过为关系定义 `InlineModelAdmin` 对象来实现：

```
from django.contrib import admin

class MembershipInline(admin.TabularInline):
    model = Group.members.through

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]

class GroupAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]
    exclude = ('members',)
```

在这个例子中有两个值得注意的特征。

首先 - `MembershipInline` 类引用 `Group.members.through` 。 `through` 属性是对管理多对多关系的模型的引用。在定义多对多字段时，此模型由Django自动创建。

其次， `GroupAdmin` 必须手动排除 `members` 字段。Django在定义关系（在这种情况下， `Group` ）的模型上显示多对多字段的管理窗口小部件。如果要使用内联模型来表示多对多关系，则必须告知Django的管理员而不是显示此窗口小部件 - 否则您最终会在管理页面上看到两个窗口小部件，用于管理关系。

在所有其他方面， `InlineModelAdmin` 与任何其他方面完全相同。您可以使用任何正常的 `ModelAdmin` 属性自定义外观。

使用多对多中介模型

当您使用 `ManyToManyField` 的 `through` 参数指定中介模型时，管理员将不会默认显示窗口小部件。这是因为该中间模型的每个实例需要比可以在单个小部件中显示的更多的信息，并且多个小部件所需的布局将根据中间模型而变化。

但是，我们仍然希望能够内联编辑该信息。幸运的是，这很容易与内联管理模型。假设我们有以下模型：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership
')

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

在管理中显示此中间模型的第一步是为 `Membership` 模型定义一个内联类：

```
class MembershipInline(admin.TabularInline):
    model = Membership
    extra = 1
```

此简单示例使用 `Membership` 模型的默认 `InlineModelAdmin` 值，并将额外添加表单限制为一个。这可以使用 `InlineModelAdmin` 类可用的任何选项进行自定义。

现在为 `Person` 和 `Group` 模型创建管理视图：

```
class PersonAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)

class GroupAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)
```

最后，向管理网站注册您的 `Person` 和 `Group` 模型：

```
admin.site.register(Person, PersonAdmin)
admin.site.register(Group, GroupAdmin)
```

现在，您的管理网站已设置为从 `Person` 或 `Group` 详细信息页面内联编辑 `Membership` 对象。

使用泛型关系作为内联

可以使用内联与一般相关的对象。假设您有以下型号：

```
from django.db import models
from django.contrib.contenttypes.fields import GenericForeignKey

class Image(models.Model):
    image = models.ImageField(upload_to="images")
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey("content_type", "object_i
d")

class Product(models.Model):
    name = models.CharField(max_length=100)
```

If you want to allow editing and creating `Image` instance on the `Product` add/change views you can use `GenericTabularInline` or `GenericStackedInline` (both subclasses of `GenericInlineModelAdmin`) provided by `admin`, they implement tabular and stacked visual layouts for the forms representing the inline objects respectively just like their non-generic counterparts and behave just like any other inline. 在此示例应用的 `admin.py` 中：

```
from django.contrib import admin
from django.contrib.contenttypes.admin import GenericTabularInli
ne

from myproject.myapp.models import Image, Product

class ImageInline(GenericTabularInline):
    model = Image

class ProductAdmin(admin.ModelAdmin):
    inlines = [
        ImageInline,
    ]

admin.site.register(Product, ProductAdmin)
```

有关更多具体信息，请参阅*contenttypes documentation*。

重写 admin 模板

相对重写一个admin站点的各类页面，直接在admin站点默认templates上直接进行修改是件相对简单的事。你甚至可以为特定的应用或一个特定的模型覆盖少量的这些模板。

设置项目的Admin模板目录

Admin模板文件位于 `contrib/admin/templates/admin` 目录中。

如要覆盖一个或多个模板，首先在你的项目的 `templates` 目录中创建一个 `admin` 目录。它可以是你在 `TEMPLATES` 设置的 `DjangoTemplates` 后端的 `DIRS` 选项中指定的任何目录。如果你已经自定义 `'loaders'` 选项，请确保 `'django.template.loaders.filesystem.Loader'` 出现在 `'django.template.loaders.app_directories` 之前。Loader'，以便在包含 `django.contrib.admin` 的模板之前，模板加载系统可以找到您的自定义模板。

在 `admin` 目录下, 以你的应用名创建子目录. 在应用名的目录下，以你模型层的名字创建子目录. 注意：admin应用会以小写名的形式在目录下查找模型, 如果你想在大小写敏感的文件系统上运行app，请确保以小写形式命名目录.

为一个特定的app重写admin模板, 需要拷贝 `django/contrib/admin/templates/admin` 目录到你刚才创建的目录下, 并且修改它们.

For example, if we wanted to add a tool to the change list view for all the models in an app named `my_app`, we would copy `contrib/admin/templates/admin/change_list.html` to the `templates/admin/my_app/` directory of our project, and make any necessary changes.

如果我们只想为名为“Page”的特定模型添加一个工具到更改列表视图，我们将把同一个文件复制到我们项目的 `templates/admin/my_app/page` 目录。

覆盖与替换管理模板

由于管理模板的模块化设计，通常既不必要也不建议替换整个模板。最好只覆盖模板中需要更改的部分。

要继续上述示例，我们要为 `Page` 模型的 `History` 工具旁边添加一个新链接。查看 `change_form.html` 后，我们确定我们只需要覆盖 `object-tools-items` 块。因此，这里是我们的新 `change_form.html` ：

```
{% extends "admin/change_form.html" %}
{% load i18n admin_urls %}
{% block object-tools-items %}
    <li>
        <a href="{% url opts|admin_urlname:'history' original.pk
|admin_urlquote %}" class="historylink">{% trans "History" %}</a
>
    </li>
    <li>
        <a href="mylink/" class="historylink">My Link</a>
    </li>
    {% if has_absolute_url %}
        <li>
            <a href="{% url 'admin:view_on_site' content_type_id
 original.pk %}" class="viewsitelink">{% trans "View on site" %}
</a>
        </li>
    {% endif %}
{% endblock %}
```

就是这样！如果我们将此文件放在 `templates/admin/my_app` 目录中，我们的链接将出现在my_app中所有模型的更改表单上。

每个应用或模型中可以被重写的模板

不是 `contrib/admin/templates/admin` 中的每个模板都可以在每个应用或每个模型中覆盖。以下可以 ：

- `app_index.html`
- `change_form.html`
- `change_list.html`
- `delete_confirmation.html`
- `object_history.html`

对于那些不能以这种方式重写的模板，你可能仍然为您的整个项目重写它们。只需要将新版本放在你的 `templates/admin` 目录下。这对于要创建自定义的404 和500 页面特别有用。

注意

一些Admin的模板，例如 `change_list_results.html` 用于呈现自定义包含标签。这些可能会被覆盖，但在这种情况下你可能最好是创建您自己的版本Tag，并给它一个不同的名称。这样你可以有选择地使用它。

Root and login 模板

如果你想要更改主页、登录或登出页面的模板，你最后创建你自己的 `AdminSite` 实例（见下文），并更改 `AdminSite.index_template` 、`AdminSite.login_template` 和 `AdminSite.logout_template` 属性。

AdminSite objects

class `AdminSite` (*name='admin'*)

Django 的一个Admin 站点通过 `django.contrib.admin.sites.AdminSite` 的一个实例表示；默认创建的这个类实例是 `django.contrib.admin.site` ，你可以通过它注册自己的模型和 `ModelAdmin` 实例。

当构造 `AdminSite` 的实例时，你可以使用 `name` 参数给构造函数提供一个唯一的实例名称。这个实例名称用于标识实例，尤其是反向解析*Admin URLs* 的时候。如果没有提供实例的名称，将使用默认的实例名称 `admin` 。有关自定义 `AdminSite` 类的示例，请参见自定义*AdminSite* 类。

AdminSite attributes

如覆盖*Admin* 模板中所述，模板可以覆盖或扩展基础的Admin 模板。

`AdminSite.``site_header`

New in Django 1.7.

每个Admin 页面顶部的文本，形式为 `<h1>` （字符串）。默认为 “Django administration”。

`AdminSite.``site_title`

New in Django 1.7.

每个Admin 页面底部的文本，形式为 `<title>` （字符串）。默认为“Django site admin”。

`AdminSite.``site_url`

New in Django 1.8.

每个Admin 页面顶部"View site" 链接的URL。默认情况下， `site_url` 为 `/` 。设置为 `None` 可以删除这个链接。

`AdminSite.``index_title`

New in Django 1.7.

Admin 主页顶部的文本（一个字符串）。默认为 “Site administration”。

`AdminSite.``index_template`

Admin 站点主页的视图使用的自定义模板的路径。

`AdminSite.``app_index_template`

Admin 站点app index 的视图使用的自定义模板的路径。

`AdminSite.``login_template`

Admin 站点登录视图使用的自定义模板的路径。

`AdminSite.``login_form`

Admin 站点登录视图使用的 `AuthenticationForm` 的子类。

`AdminSite.``logout_template`

Admin 站点登出视图使用的自定义模板的路径。

`AdminSite.``password_change_template`

Admin 站点密码修改视图使用的自定义模板的路径。

`AdminSite.``password_change_done_template`

Admin 站点密码修改完成视图使用的自定义模板的路径。

AdminSite methods

`AdminSite.``each_context` (*request*)

New in Django 1.7.

返回一个字典，包含将放置在Admin 站点每个页面的模板上下文中的变量。

包含以下变量和默认值：

- `site_header` ：`AdminSite.site_header`
- `site_title` ：`AdminSite.site_title`
- `site_url` ：`AdminSite.site_url`
- `has_permission` ：`AdminSite.has_permission()`

Changed in Django 1.8:

添加 `request` 参数和 `has_permission` 变量。

`AdminSite.``has_permission` (*request*)

对于给定的 `HttpRequest` ，如果用户有权查看Admin 网站中的至少一个页面，则返回 `True` 。默认要求 `User.is_active` 和 `User.is_staff` 都为 `True` 。

绑定 `AdminSite`

设置Django Admin 的最后一步是放置你的 `AdminSite` 到你的URLconf 中。通过指向给定的URL 到 `AdminSite.urls` 方法来执行此操作。

在下面的示例中，我们注册默认的 `AdminSite` 实例 `django.contrib.admin.site` 到 URL `/admin/` 。

```
# urls.py
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
]
```

自定义 `AdminSite`

如果你想要建立你自己的具有自定义行为Admin 站点，你可以自由地子类化 `AdminSite` 并重写或添加任何你喜欢的东西。你只需创建 `AdminSite` 子类的实例（方式与你会实例化任何其它Python 类相同） 并注册你的模型和 `ModelAdmin` 子类与它而不是默认的站点。最后，更新 `myproject/urls.py` 来引用你的 `AdminSite` 子类。

myapp/admin.py

```
from django.contrib.admin import AdminSite

from .models import MyModel

class MyAdminSite(AdminSite):
    site_header = 'Monty Python administration'

admin_site = MyAdminSite(name='myadmin')
admin_site.register(MyModel)
```

myproject/urls.py

```
from django.conf.urls import include, url

from myapp.admin import admin_site

urlpatterns = [
    url(r'^myadmin/', include(admin_site.urls)),
]
```

注意，当使用你自己的 `AdminSite` 实例时，你可能不希望自动发现 `admin` 模块，因为这将导入 `admin` 模块到你的每个 `myproject.admin` 模块中 。这时，你需要将 `'django.contrib.admin.apps.SimpleAdminConfig'` 而不是 `'django.contrib.admin'` 放置在你的 `INSTALLED_APPS` 设置中。

相同URLconf 中有多个Admin 站点

在Django 构建的同一Web 站点上创建Admin 站点的多个实例非常容易。只需要创建 `AdminSite` 的多个实例并将每个实例放置在不同的URL 下。

在下面的示例中， `/basic-admin/` 和 `/advanced-admin/` 分别使用 `AdminSite` 的 `myproject.admin.basic_site` 实例和 `myproject.admin.advanced_site` 实例表示不同版本的Admin 站点：

```
# urls.py
from django.conf.urls import include, url
from myproject.admin import basic_site, advanced_site

urlpatterns = [
    url(r'^basic-admin/', include(basic_site.urls)),
    url(r'^advanced-admin/', include(advanced_site.urls)),
]
```

`AdminSite` 实例的构造函数中接受一个单一参数用做它们的名字，可以是任何你喜欢的东西。此参数将成为反向解析它们 时URL 名称的前缀。只有在你使用多个 `AdminSite` 时它才是必要的。

向管理网站添加视图

与 `ModelAdmin` 一样， `AdminSite` 提供了一个 `get_urls()` 方法，可以重写该方法以定义网站的其他视图。要向您的管理网站添加新视图，请扩展基本 `get_urls()` 方法，为新视图添加模式。

注意

您呈现的任何使用管理模板的视图或扩展基本管理模板，应在渲染模板之前设置 `request.current_app` 。It should be set to either `self.name` if your view is on an `AdminSite` or `self.admin_site.name` if your view is on a `ModelAdmin` .

Changed in Django 1.8:

在以前的Django版本中，在呈现模板时，您必须向 `RequestContext` 或 `Context` 提供 `current_app` 参数。

加入一个密码重置的特性

想admin site加入密码重置功能只需要在url配置文件中简单加入几行代码即可。具体操作就是加入下面四个正则规则。

```
from django.contrib.auth import views as auth_views

url(r'^admin/password_reset/$', auth_views.password_reset, name=
'admin_password_reset'),
url(r'^admin/password_reset/done/$', auth_views.password_reset_d
one, name='password_reset_done'),
url(r'^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>.+)/$', auth_
views.password_reset_confirm, name='password_reset_confirm'),
url(r'^reset/done/$', auth_views.password_reset_complete, name='
password_reset_complete'),
```

（假设您已在 `admin/` 添加了管理员，并要求您在包含管理应用程序的行之前将 `^admin/` 开头的网址）。

如果存在 `admin_password_reset` 命名的URL，则会在密码框下的默认管理登录页面上显示“忘记了您的密码？”链接。

反向解析Admin 的URL

`AdminSite` 部署后，该站点所提供的视图都可以使用Django 的*URL 反向解析系统*访问。

`AdminSite` 提供以下命名URL：

Page	URL name	Parameters
Index	`index`	
Logout	`logout`	
Password change	`password_change`	
Password change done	`password_change_done`	
i18n JavaScript	`jsi18n`	
Application index page	`app_list`	`app_label`
Redirect to object's page	`view_on_site`	`content_type_id`, `object_id`

每个 `ModelAdmin` 实例还将提供额外的命名URL：

Page	URL name	Parameters
Changelist	`__changelist`	
Add	`__add`	
History	`__history`	`object_id`
Delete	`__delete`	`object_id`
Change	`__change`	`object_id`

这些命名URL 注册的应用命名空间为 `admin` ，实例命名空间为对应的AdminSite 实例的名称。

所以，如果你想要获取默认Admin 中，（polls 应用的） 一个特定的 `Choice` 对象的更改视图的引用，你可以调用：

```
>>> from django.core import urlresolvers
>>> c = Choice.objects.get(...)
>>> change_url = urlresolvers.reverse('admin:polls_choice_change
', args=(c.id,))
```

这将找到管理应用程序的第一个注册实例（无论实例名称），并解析到视图以更改 `poll.` 该实例中的选择实例。

如果你想要查找一个特定的Admin 实例中URL，请提供实例的名称作为 `current_app` 给反向解析的调用 。例如，如果你希望得到名为 `custom` 的Admin 实例中的视图，你将需要调用：

```
>>> change_url = urlresolvers.reverse('admin:polls_choice_change
',
...                                     args=(c.id,), current_app=
'custom')
```

有关更多详细信息，请参阅反向解析名称空间*URL* 的文档。

为了让模板中反向解析Admin URL 更加容易，Django 提供一个 `admin_urlname` 过滤器，它以Action 作为参数：

```
{% load admin_urls %}
<a href="{% url opts|admin_urlname:'add' %}">Add user</a>
<a href="{% url opts|admin_urlname:'delete' user.pk %}">Delete t
his user</a>
```

在上面的例子中Action 将匹配上文所述的 `ModelAdmin` 实例的URL 名称的最后部分。 `opts` 变量可以是任何具有 `app_label` 和 `model_name` 属性的对象，通常由Admin 视图为当前的模型提供。

管理操作

简而言之，Django管理后台的基本流程是，“选择一个对象并改变它”。在大多数情况下，这是非常适合的。然而当你一次性要对多个对象做相同的改变，这个流程是非常的单调乏味的。

在这些例子中，Django管理后台可以让你实现和注册“操作”—— 仅仅只是一个以已选中对象集合为参数的回调函数。

在Django自带的管理页面中都能看到这样的例子。Django在所有的模型中自带了一个“删除所选对象”操作。例如，下面是 `django.contrib.auth` app 在Django's创建的用户模型:

Select user to change

Search

Action: --------- Go 1 of 3 selected

	Username	E-mail address	First name	Last name	Staff status
☐	adrian	adrian@example.com	Adrian	Holovaty	
☑	jacob	jacob@example.com	Jacob	Kaplan-Moss	
☐	simon	simon@example.com	Simon	Willison	

3 users

警告

“删除所选对象”的操作由于性能因素使用了 `QuerySet.delete()` ，这里有个附加说明：它不会调用你模型的 `delete()` 方法。

如果你想覆写这一行为，编写自定义操作，以你的方式实现删除就可以了 -- 例如，对每个已选择的元素调用 `Model.delete()` 。

关于整体删除的更多信息，参见对象删除的文档。

继续阅读，来弄清楚如何向列表添加你自己的操作。

编写操作

通过示例来解释操作最为简单，让我们开始吧。

操作的一个最为普遍的用例是模型的整体更新。考虑带有 `Article` 模型的简单新闻应用：

```
from django.db import models

STATUS_CHOICES = (
    ('d', 'Draft'),
    ('p', 'Published'),
    ('w', 'Withdrawn'),
)

class Article(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()
    status = models.CharField(max_length=1, choices=STATUS_CHOIC
ES)

    def __str__(self):              # __unicode__ on Python 2
        return self.title
```

我们可能在模型上执行的一个普遍任务是，将文章状态从“草稿”更新为“已发布”。我们在后台一次处理一篇文章非常轻松，但是如果我们想要批量发布一些文章，会非常麻烦。所以让我们编写一个操作，可以让我们将一篇文章的状态修改为“已发布”。

编写操作 函数

首先，我们需要定义一个函数，当后台操作被点击触发的时候调用。操作函数，跟普通的函数一样，需要接收三个参数：

- 当前的 `ModelAdmin`
- 表示当前请求的 `HttpRequest`
- 含有用户所选的对象集合的 `QuerySet`

我们用于发布这些文章的函数并不需要 `ModelAdmin` 或者请求对象，但是我们会用到查询集：

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
```

注意

为了性能最优，我们使用查询集的*update 方法*。其它类型的操作可能需要分别处理每个对象；这种情况下我们需要对查询集进行遍历：

```
for obj in queryset:
    do_something_with(obj)
```

编写操作的全部内容实际上就这么多了。但是，我们要进行一个可选但是有用的步骤，在后台给操作起一个“非常棒”的标题。通常，操作以“Make published”的方式出现在操作列表中 -- 所有空格被下划线替换后的函数名称。这样就很好了，但是我们可以提供一个更好、更人性化的名称，通过向 `make_published` 函数添加 `short_description` 属性：

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"
```

注意

这看起来可能会有点熟悉；admin的 `list_display` 选项使用同样的技巧，为这里注册的回掉函数来提供人类可读的描述。

添加操作到 `ModelAdmin`

接下来，我们需要把操作告诉 `ModelAdmin` 。它和其他配置项的工作方式相同。所以，带有操作及其注册的完整的 `admin.py` 看起来像这样：

```
from django.contrib import admin
from myapp.models import Article

def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"

class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'status']
    ordering = ['title']
    actions = [make_published]

admin.site.register(Article, ArticleAdmin)
```

这段代码会向我们提供admin的更改列表，看起来像这样：

这就是全部内容了。如果你想编写自己的操作，你现在应该知道怎么开始了。这篇文档的剩余部分会介绍更多高级技巧。

在操作中处理错误

如果你预见到，运行你的操作时可能出现一些错误，你应该以优雅的方式向用户通知这些错误。也就是说，异常处理和使用 `django.contrib.admin.ModelAdmin.message_user()` 可以在响应中展示用户友好的问题描述。

操作的高级技巧

对于进一步的选择，你可以使用一些额外的选项。

ModelAdmin上的操作 `ModelAdmin`

上面的例子展示了定义为一个简单函数的 `make_published` 操作。这真是极好的，但是以视图的代码设计角度来看，它并不完美：由于操作与 `Article` 紧密耦合，不如将操作直接绑定到 `ArticleAdmin` 对象上更有意义。

这样做十分简单：

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    actions = ['make_published']

    def make_published(self, request, queryset):
        queryset.update(status='p')
    make_published.short_description = "Mark selected stories as published"
```

首先注意，我们将 `make_published` 放到一个方法中，并重命名 `modeladmin` 为 `self` ，其次，我们现在将 `'make_published'` 字符串放进了 `actions` ，而不是一个直接的函数引用。这样会让 `ModelAdmin` 将这个操作视为方法。

将操作定义为方法，可以使操作以更加直接、符合语言习惯的方式来访问 `ModelAdmin` ，调用任何admin提供的方法。

例如，我们可以使用 `self` 来向用户发送消息，告诉她操作成功了：

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    def make_published(self, request, queryset):
        rows_updated = queryset.update(status='p')
        if rows_updated == 1:
            message_bit = "1 story was"
        else:
            message_bit = "%s stories were" % rows_updated
        self.message_user(request, "%s successfully marked as pu
blished." % message_bit)
```

这会使动作与后台在成功执行动作后做的事情相匹配：



提供中间页面的操作

通常，在执行操作之后，用户会简单地通过重定向返回到之前的修改列表页面中。然而，一些操作，尤其是更加复杂的操作，需要返回一个中间页面。例如，内建的删除操作，在删除选中对象之前需要向用户询问来确认。

要提供中间页面，只要从你的操作返回 `HttpResponse` （或其子类）就可以了。例如，你可能编写了一个简单的导出函数，它使用了Django的序列化函数来将一些选中的对象转换为JSON：

```
from django.http import HttpResponse
from django.core import serializers

def export_as_json(modeladmin, request, queryset):
    response = HttpResponse(content_type="application/json")
    serializers.serialize("json", queryset, stream=response)
    return response
```

通常，上面的代码的实现方式并不是很好。大多数情况下，最佳实践是返回 `HttpResponseRedirect` ，并且使用户重定向到你编写的视图中，向GET查询字符串传递选中对象的列表。这需要你在中间界面上提供复杂的交互逻辑。例如，如果你打算提供一个更加复杂的导出函数，你会希望让用户选择一种格式，以及可能在导出中包含一个含有字段的列表。最佳方式是编写一个小型的操作，简单重定向到你的自定义导出视图中：

```
from django.contrib import admin
from django.contrib.contenttypes.models import ContentType
from django.http import HttpResponseRedirect

def export_selected_objects(modeladmin, request, queryset):
    selected = request.POST.getlist(admin.ACTION_CHECKBOX_NAME)
    ct = ContentType.objects.get_for_model(queryset.model)
    return HttpResponseRedirect("/export/?ct=%s&ids=%s" % (ct.pk
, ",".join(selected)))
```

就像你看到的那样，这个操作是最简单的部分；所有复杂的逻辑都在你的导出视图里面。这需要处理任何类型的对象，所以需要处理 `ContentType` 。

这个视图的编写作为一个练习留给读者。

在整个站点应用操作

`AdminSite.``add_action` (*action*[, *name*])

如果一些操作对管理站点的任何对象都可用的话，是非常不错的 -- 上面所定义的导出操作是个不错的备选方案。你可以使用 `AdminSite.add_action()` 让一个操作在全局都可以使用。例如：

```
from django.contrib import admin

admin.site.add_action(export_selected_objects)
```

这样， `export_selected_objects` 操作可以在全局使用，名称为“export*selected_objects*”。你也可以显式指定操作的名称 – 如果你想以编程的方式*[移除这个操作_*](#disabling-admin-actions) – 通过向 `AdminSite.add_action()` 传递第二个参数：

```
admin.site.add_action(export_selected_objects, 'export_selected'
)
```

禁用操作

有时你需要禁用特定的操作 -- 尤其是注册的站点级操作 -- 对于特定的对象。你可以使用一些方法来禁用操作：

禁用整个站点的操作

`AdminSite.``disable_action` (*name*)

如果你需要禁用站点级操作 ，你可以调用 `AdminSite.disable_action()` 。

例如，你可以使用这个方法来移除内建的“删除选中的对象”操作：

```
admin.site.disable_action('delete_selected')
```

一旦你执行了上面的代码，这个操作不再对整个站点中可用。

然而，如果你需要为特定的模型重新启动在全局禁用的对象，把它显式放在 `ModelAdmin.actions` 列表中就可以了：

```
# Globally disable delete selected
admin.site.disable_action('delete_selected')

# This ModelAdmin will not have delete_selected available
class SomeModelAdmin(admin.ModelAdmin):
    actions = ['some_other_action']
    ...

# This one will
class AnotherModelAdmin(admin.ModelAdmin):
    actions = ['delete_selected', 'a_third_action']
    ...
```

为特定的ModelAdmin禁用所有操作 `ModelAdmin`

如果你想批量移除所提供 `ModelAdmin` 上的所有操作，可以把 `ModelAdmin.actions` 设置为 `None` ：

```
class MyModelAdmin(admin.ModelAdmin):
    actions = None
```

这样会告诉 `ModelAdmin` ，不要展示或者允许任何操作，包括站点级操作。

按需启用或禁用操作

`ModelAdmin.``get_actions` (*request*)

最后，你可以通过覆写 `ModelAdmin.get_actions()` ，对每个请求（每个用户）按需开启或禁用操作。

这个函数返回包含允许操作的字典。字典的键是操作的名称，值是 `(function, name, short_description)` 元组。

多数情况下，你会按需使用这一方法，来从超类中的列表移除操作。例如，如果我只希望名称以'J'开头的用户可以批量删除对象，我可以执行下面的代码：

```
class MyModelAdmin(admin.ModelAdmin):
    ...

    def get_actions(self, request):
        actions = super(MyModelAdmin, self).get_actions(request)
        if request.user.username[0].upper() != 'J':
            if 'delete_selected' in actions:
                del actions['delete_selected']
        return actions
```

> 译者：Django 文档协作翻译小组，原文：Admin actions。
>
>
>
>

Django管理文档生成器

Django的 `admindocs` 应用从模型、视图、模板标签以及模板过滤器中，为任何 `INSTALLED_APPS` 中的应用获取文档。并且让文档可以在 `Django admin` 中使用。

在某种程度上，你可以使用 `admindocs` 来快为你自己的代码生成文档。这个应用的功能十分有限，然而它主要用于文档模板、模板标签和过滤器。例如，需要参数的模型方法在文档中会有意地忽略，因为它们不能从模板中调用。这个应用仍旧有用，因为它并不需要你编写任何额外的文档（除了 `docstrings` ），并且在 `Django admin` 中使用很方便。

概览

要启用 `admindocs` ，你需要执行以下步骤：

- 向 `INSTALLED_APPS` 添加 `django.contrib.admindocs` 。
- 向你的 `urlpatterns` 添加(`r'^admin/doc/'` , `include('django.contrib.admindocs.urls')`)。 确保它在 `r'^admin/'` 这一项 之前包含，以便 `/admin/doc/` 的请求不会被后面的项目处理。
- 安装 `docutils` Python 模块 (http://docutils.sf.net/)。
- 可选的： 使用 `admindocs` 的书签功能需要安装 `django.contrib.admindocs.middleware.XViewMiddleware` 。

一旦完成这些步骤，你可以开始通过你的 `admin` 接口和点击在页面右上方的“Documentation”链接来浏览文档。

文档助手

下列特定的标记可以用于你的 `docstrings` ，来轻易创建到其他组件的超链接：

Django Component	reStructuredText roles
Models	:model:`app_label.ModelName`
Views	:view:`app_label.view_name`
Template tags	:tag:`tagname`
Template filters	:filter:`filtername`
Templates	:template:`path/to/template.html`

模型参考

`admindocs` 页面的 `models` 部分描述了系统中每个模型，以及所有可用的字段和方法（不带任何参数）。虽然模型的属性没有任何参数，但他们没有列出。和其它模型的关联以超链接形式出现。描述由字段上的 `help_text` 属性，或者从模型方法的 `docstrings` 导出。

带有有用文档的模型看起来像是这样：

```
class BlogEntry(models.Model):
    """
    Stores a single blog entry, related to :model:`blog.Blog` an
d
    :model:`auth.User`.

    """
    slug = models.SlugField(help_text="A short label, generally
used in URLs.")
    author = models.ForeignKey(User)
    blog = models.ForeignKey(Blog)
    ...

    def publish(self):
        """Makes the blog entry live on the site."""
        ...
```

视图参考

你站点中的每个URL都在·页面中有一个单独的记录，点击提供的URL会向你展示相应的视图。有一些有用的东西，你可以在你的视图函数的·中记录：

- 视图所做工作的一个简短的描述。
- 上下文，或者是视图的模板中可用变量的列表。
- 用于当前视图的模板的名称。

例如：

```
from django.shortcuts import render

from myapp.models import MyModel

def my_view(request, slug):
    """
    Display an individual :model:`myapp.MyModel`.

    **Context**

    ``mymodel``
        An instance of :model:`myapp.MyModel`.

    **Template:**

    :template:`myapp/my_template.html`

    """
    context = {'mymodel': MyModel.objects.get(slug=slug)}
    return render(request, 'myapp/my_template.html', context)
```

模板标签和过滤器参考

`admindocs` 的 `tags` 和 `filters` 部分描述了Django自带的所有标签和过滤器（事实上，内建的标签参考 和 内建的过滤器参考文档直接来自于那些页面）。你创建的，或者由三方应用添加的任何标签或者过滤器，也会在这一部分中展示。

模板参考

虽然 `admindocs` 并不包含一个地方来保存模板，但如果你在结果页面中使用 `:template:`path/to/template.html`` 语法，会使用Django的模板加载器来验证该模板的路径。这是一个非常便捷的方法，来检查是否存在特定的模板，以及展示模板在文件系统的何处存放。

包含的书签

`admindocs` 页面上有一些很有用的书签：

Documentation for this page

Jumps you from any page to the documentation for the view that generates that page.

Show object ID

Shows the content-type and unique ID for pages that represent a single object.

Edit this object

Jumps to the admin page for pages that represent a single object.

为使用这些书签，你需要用带有 `is_staff` 设置为 `True` 的 `User` 登录 `Django admin` ，或者安装了 `XViewMiddleware` 并且你通过 `INTERNAL_IPS` 中的IP地址访问站点。

> 译者：Django 文档协作翻译小组，原文：Admin documentation generator。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

安全

安全在Web应用开发中是一项至关重要的话题，Django提供了多种保护手段和机制：

Django安全

这份文档是 Django 的安全功能的概述。 它包括给 Django 驱动的网站一些加固建议。

跨站脚本 (XSS) 防护

XSS攻击允许用户注入客户端脚本到其他用户的浏览器里。 这通常是通过存储在数据库中的恶意脚本，它将检索并显示给其他用户，或者通过让用户点击一个链接，这将导致攻击者的 JavaScript 被用户的浏览器执行。 然而，XSS 攻击可以来自任何不受信任的源数据，如 Cookie 或 Web 服务，任何没有经过充分处理就包含在网页中的数据。

使用 Django 模板保护你免受多数 XSS 攻击。 然而，重要的是要了解它提供了什么保护及其局限性。

Django 模板会 编码特殊字符 ，这些字符在 HTML 中都是特别危险的。 虽然这可以防止大多数恶意输入的用户，但它不能完全保证万无一失。 例如，它不会防护以下内容：

```
<style class=>...</style>
```

如果 `var` 设置为 `'class1 onmouseover=javascript:func()'`，这可能会导致在未经授权的 JavaScript 的执行，取决于浏览器如何呈现不完整的 HTML。 （对属性值使用引号可以修复这种情况。）

同样重要的是`is_safe` 要特别小心的用在 自定义模板标签，`safe` 模板标签，`mark_safe` ，还有 autoescape 被关闭的时候。

此外，如果您使用的是模板系统输出 HTML 以外的东西，可能会有完全不同的字符和单词需要编码。

你也应该在数据库中存储 HTML 的时候要非常小心，尤其是当 HTML 被检索然后展示出来。

跨站请求伪造 (CSRF) 防护

CSRF 攻击允许恶意用户在另一个用户不知情或者未同意的情况下，以他的身份执行操作。

Django 对大多数类型的 CSRF 攻击有内置的保护，在适当情况下你可以开启并使用它 。 然而，对于任何解决技术，都有它的局限性。 例如，CSRF 模块可以在全局范围内或为特定视图被禁用 。 您应该只在您知道在做什么的情况下操作。 还有其他 限制 如果你的网站有子域名并且在你的控制之外。

CSRF 防护 是通过检查每个 POST 请求的一个随机数（nonce）来工作。 这确保了恶意用户不能简单"回放"你网站上面表单的POST，以及让另一个登录的用户无意中提交表单。恶意用户必须知道这个随机数，它是用户特定的（存在cookie里）。

使用 HTTPS来部署的时候， `CsrfViewMiddleware` 会检查HTTP referer协议头是否设置为同源的URL（包括子域和端口）。因为HTTPS提供了附加的安全保护，转发不安全的连接请求时，必须确保链接使用 HTTPS，并使用HSTS支持的浏览器。

使用 `csrf_exempt` 装饰器来标记视图时，要非常小心，除非这是极其必要的。

SQL 注入保护

SQl注入是一种攻击类型，恶意用户可以在系统数据库中执行任意SQL代码。这可能会导致记录删除或者数据泄露。

通过使用Django的查询集，产生的SQL会由底层数据库驱动正确地转义。然而，Django也允许开发者编写原始查询或者执行自定义sql。这些功能应该谨慎使用，并且你应该时刻小心正确转义任何用户可以控制的参数。另外，你在使用 `extra()` 的时候应该谨慎行事。

点击劫持保护

点击劫持是一类攻击，恶意站点在一个frame中包裹了另一个站点。这类攻击可能导致用户被诱导在目标站点做出一些无意识的行为。

Django在 `X-Frame-Options` 中间件 的表单中中含有 点击劫持保护 ，它在支持的浏览器中可以保护站点免于在frame中渲染。也可以在每个视图中禁止这一保护，或者配置要发送的额外的协议头。

对于任何不需要将页面包装在三方站点的frame中，或者只需要包含它的一部分的站点，都强烈推荐启用这一中间件。

SSL / HTTPS

把你的站点部署在HTTPS下总是更安全的，尽管不是在所有情况下都有效。如果不这样，恶意的网络用户可能会嗅探授权证书，或者其他在客户端和服务端之间传输的信息，或者一些情况下 -- 活跃的网络攻击者 -- 会修改在两边传输的数据。

如果你想要HTTPS提供保护，那么需要在你的服务器上启用它，可能还需要去做一些额外的操作

- 如果必要的话，设置 SECURE_PROXY_SSL_HEADER ，确保你已经彻底了解警告。未能实现它会导致CSRF方面的缺陷，也是很危险的！
- 设置重定向，那样通过HTTP的请求会重定向到HTTPS。

这可以通过自定义的中间件来实现。请注意 `SECURE_PROXY_SSL_HEADER` 下的警告。对于反向代理的情况，配置web主服务器来重定向到HTTPS或许是最简单也许是最安全的做法。

- 使用“安全的”cookie。

  如果浏览器的连接一开始通过HTTP，这是大多数浏览器的通常情况，已存在的cookie可能会被泄露。因此，你应该将 `SESSION_COOKIE_SECURE` 和 `CSRF_COOKIE_SECURE` 设置为 `True` 。这会使浏览器只在HTTPS连接中发送这些cookie。要注意这意味着会话在HTTP下不能工作，并且CSRF保护功能会在HTTP下阻止接受任何POST数据（如果你把所有HTTP请求都重定向到HTTPS之后就没问题了）。

- 使用 HTTP 强制安全传输 (HSTS)

  HSTS 是一个HTTP协议头，它通知浏览器，到特定站点的所有链接都一直使用HTTPS。通过和重定向HTTP请求到HTTPS一起使用，确保连接总是享有附加的SSL安全保障，由一个已存在的成功的连接提供。HSTS通常在web服务器上面配置。

Host 协议头验证

在某些情况下，Django使用客户端提供的 `Host` 协议头来构造URL。虽然这些值可以被审查，来防止跨站脚本攻击（XSS），但是一个假的 `Host` 值可以用于跨站请求伪造（CSRF），有害的缓存攻击，以及email中的有害链接。

因为即使表面上看起来安全的web服务器也容易被篡改 `主机头` ，Django再次在django.http.HttpRequest.get_host()这个方法中验证主机头这个ALLOWED_HOSTS的设置 Django validates `Host` headers against the `ALLOWED_HOSTS` setting in the `django.http.HttpRequest.get_host()` method.

验证只通过 `get_host()` 来应用；如果你的代码从 `request.META` 中直接访问 `Host` 协议头，就会绕开这一安全防护。

详见完整的 `ALLOWED_HOSTS` 文档。

警告

本文档的先前版本建议配置Web服务器以确保其验证传入的HTTP `Host` 头。虽然这仍然是建议，在许多常见的Web服务器，似乎验证 `Host` 头的配置可能实际上不这样做。例如，即使Apache配置为使您的Django站点从设置了 `ServerName` 的非默认虚拟主机提供，HTTP请求仍然可以匹配此虚拟主机并提供假 `Host` 标头。因此，Django现在要求您明确设置 `ALLOWED_HOSTS` ，而不是依赖于Web服务器配置。

另外，就像1.3.1，如果你的配置需要它的话，Django 需要你显式开启对 `X-Forwarded-Host` 协议头的支持(通过 `USE_X_FORWARDED_HOST` 设置)。

Session 会话安全

类似于部署在站点上的*CSRF 限制* 使不受信任的用户不能访问任何子域， `django.contrib.sessions` 也有一些限制。详见安全中会话的话题指南。

用户上传的内容

注意

考虑在云服务器或*CDN*上面部署静态文件来避免一些此类问题。

- 如果你的站点接受上传文件，强烈推荐你在web服务器配置中，将这些上传限制为合理的大小，来避免拒绝服务（DOS）攻击。在Apache中，这可以简单地使用LimitRequestBody指令。
- 如果你自己处理静态文件，确保像Apache的 `mod_php` 的处理器已关闭，它会将静态文件执行为代码。你并不希望用户能够通过上传和请求一个精心构造的文件来执行任意代码。
- 当媒体以不遵循安全最佳做法的方式提供时，Django的媒体上传处理带来一些漏洞。具体来说，如果HTML文件包含有效的PNG标头，然后是恶意HTML，则可以将其上传为图片。此文件将通过对Django用于 `ImageField` 图像处理（Pillow）的库的验证。当此文件随后显示给用户时，可能会显示为HTML，具体取决于Web服务器的类型和配置。

  在框架级别上没有安全验证所有用户上传的文件内容的防弹技术解决方案，但是，还可以采取一些其他步骤来减轻这些攻击：

  1. 通过始终提供来自不同顶级或二级域的用户上传的内容，可以防止一类攻击。这可以防止受到同源策略保护（例如跨站点脚本）阻止的任何漏洞利用。For example, if your site runs on `example.com` , you would want to serve uploaded content (the `MEDIA_URL` setting) from something like `usercontent-example.com` . 不足以提供来自 `usercontent.example.com` 等子网域的内容。
  2. 除此之外，应用可以选择为用户上传的文件定义一个允许的文件扩展名的白名单，并且配置web服务器直处理这些文件。

额外的安全话题

虽然Django提供了开箱即用的，良好的安全保护，但是合理地部署你的应用，以及利用web服务器、操作系统和其他组件的安全保护仍然很重要。

- 确保你的Python代码在web服务器的根目录外。这会确保你的Python代码不会意外被解析为纯文本（或者意外被执行）。
- 小心处理任何用户上传的文件。
- Django并不限制验证用户的请求。要保护对验证系统的暴力破解攻击，你可以考虑部署一个DJango的插件或者web服务器模块来限制这些请求。

- 秘密保存 `SECRET_KEY` 。
- 使用防火墙来限制缓存系统和数据库的访问是个好主意。

安全问题归档

Django的开发小组坚定地承诺，为报告和公开安全相关问题负责，这在*Django的安全问题*中列出。

作为承诺的一部分，我们保留了下面的问题的历史列表，这些问题已经被解决和公开。对于每个问题，下面的列表包含日期、简短的描述、CVE 标识符、受影响的版本列表、完整的页面链接以及相应补丁的连接。

有一些重要的附加说明：

- 列出的受影响版本只包含了在漏洞公开时期的Django稳定的安全支持发行版。这意味着，老的版本（安全支持已经过期），以及预发行版本（alpha/beta/RC）在漏洞公开的时期也可能会受影响，但是没有列出。
- Django项目偶尔会发布安全公告，指出潜在的安全问题，可能会由不合理的配置或其他Django本身以外的问题产生。这些公告中有一些收到了CVE；这种情况下，它们会在这里列出来，但是没有任何附加的补丁或者发行版，只有描述、公开信息和CVE。

Issues prior to Django's security process

一些安全问题在Django具有规范化的安全处理流程之前被修复。对于这些问题，可能不会发布新的发行版，也不会分配CVE。

August 16, 2006 - CVE-2007-0404

CVE-2007-0404: 翻译框架中的文件名验证问题。Full description

Versions affected

- Django 0.90 (patch)
- Django 0.91 (patch)
- Django 0.95 (patch) (released January 21 2007)

January 21, 2007 - CVE-2007-0405

CVE-2007-0405: 已认证用户的可见"缓存"。Full description

Versions affected

- Django 0.95 (patch)

Issues under Django’s security process

所有其它的安全问题都已经在Django安全处理流程下的版本中解决。下面会列出来：

October 26, 2007 - CVE-2007-5712

CVE-2007-5712: 通过任意大尺寸 `Accept-Language` 协议头的拒绝服务攻击。Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

May 14, 2008 - CVE-2008-2302

CVE-2008-2302: 通过admin登录重定向的XSS。Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

September 2, 2008 - CVE-2008-3909

CVE-2008-3909: 通过在admin登录状态下保存POST数据的CSRF。Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

July 28, 2009 - CVE-2009-2659

CVE-2009-2659: 开发服务器的媒体处理器上的拒绝服务攻击。Full description

Versions affected

- Django 0.96 (patch)

- Django 1.0 (patch)

October 9, 2009 - CVE-2009-3965

CVE-2009-3965: 通过执行异常正则表达式的拒绝服务攻击。Full description

Versions affected

- Django 1.0 (patch)
- Django 1.1 (patch)

September 8, 2010 - CVE-2010-3082

CVE-2010-3082: 通过不安全cookie值的XSS。Full description

Versions affected

- Django 1.2 (patch)

December 22, 2010 - CVE-2010-4534

CVE-2010-4534: 管理界面上的信息泄露。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

December 22, 2010 - CVE-2010-4535

CVE-2010-4535: 密码重置机制上的拒绝服务攻击。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0696

CVE-2011-0696: 通过伪造HTTP协议头的XSS。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0697

CVE-2011-0697: 通过未检查的名称或者上传文件的XSS。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0698

CVE-2011-0698: Windows上通过不正确的目录分隔符处理的目录遍历。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

September 9, 2011 - CVE-2011-4136

CVE-2011-4136:使用memory-cache-backed会话时的会话操纵。Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4137

CVE-2011-4137: 通过 `URLField.verify_exists` 的拒绝服务攻击。Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4138

CVE-2011-4138: 通过 `URLField.verify_exists` 的信息泄露/任何请求发布。Full description

Versions affected

- Django 1.2: (patch)
- Django 1.3: (patch)

September 9, 2011 - CVE-2011-4139

CVE-2011-4139: `Host` 协议头缓存污染。 Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4140

CVE-2011-4140:通过 `Host` 协议头的潜在CSRF威胁。Full description

Versions affected

这个通知只是一个公告，没有任何补丁发布。

- Django 1.2
- Django 1.3

July 30, 2012 - CVE-2012-3442

CVE-2012-3442: 通过验证重定向模式失败的XSS。Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

July 30, 2012 - CVE-2012-3443

CVE-2012-3443: 通过压缩的图像文件的拒绝服务u攻击。Full description

Versions affected

- Django 1.3: (patch)

- Django 1.4: (patch)

July 30, 2012 - CVE-2012-3444

CVE-2012-3444:通过大尺寸图像文件的拒绝服务攻击。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

October 17, 2012 - CVE-2012-4520

CVE-2012-4520: `Host` 协议头污染。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

December 10, 2012 - No CVE 1

对 `Host` 协议头处理的额外加固。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

December 10, 2012 - No CVE 2

对重定向验证的额外加固。Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

February 19, 2013 - No CVE

对 `Host` 协议头处理的额外加固。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-1664/1665

CVE-2013-1664 and CVE-2013-1665: 对Python XML库的基于实体的攻击。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-0305

CVE-2013-0305: 通过admin历史记录的信息泄露。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-0306

CVE-2013-0306: 通过表单集 `max_num` 的拒绝服务攻击。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

August 13, 2013 - Awaiting CVE 1

(CVE not yet issued): 通过admin受信任的 `URLField` 值的XSS。Full description

Versions affected

- Django 1.5 (patch)

August 13, 2013 - Awaiting CVE 2

(CVE not yet issued):可能的XSS漏洞，通过未验证的URL重定向模式。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

September 10, 2013 - CVE-2013-4315

CVE-2013-4315 通过 `ssi` 模板标签的目录遍历。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

September 14, 2013 - CVE-2013-1443

CVE-2013-1443: 通过长密码的拒绝服务攻击。Full description

Versions affected

- Django 1.4 (patch and Python compatibility fix)
- Django 1.5 (patch)

April 21, 2014 - CVE-2014-0472

CVE-2014-0472: 使用 `reverse()` 的非预期代码执行。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

April 21, 2014 - CVE-2014-0473

CVE-2014-0473: 匿名页面的缓存可能会泄露CSRF标识。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

April 21, 2014 - CVE-2014-0474

CVE-2014-0474: MySQL类型转换产生非预期的查询结果。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

May 18, 2014 - CVE-2014-1418

CVE-2014-1418: 缓存可能允许存储和处理私人数据。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

May 18, 2014 - CVE-2014-3730

CVE-2014-3730: 来源于用户输入的错误格式URL的不正确验证。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0480

CVE-2014-0480: reverse() 可能会生成指向其它域名的URL。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0481

CVE-2014-0481: 文件上传的拒绝服务攻击。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0482

CVE-2014-0482: RemoteUserMiddleware会话劫持。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0483

CVE-2014-0483: admin中查询集操作产生的数据泄露。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0219

CVE-2015-0219: 通过下划线或者破折号合并产生的WSGI协议头欺骗。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0220

CVE-2015-0220: 通过用户提供的重定向URL的可能的XSS攻击。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0221

CVE-2015-0221: `django.views.static.serve()` 上的拒绝服务攻击。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0222

CVE-2015-0222: 使用 `ModelMultipleChoiceField` 的数据库拒绝服务攻击。Full description

Versions affected

- Django 1.6 (patch)
- Django 1.7 (patch)

> 译者：Django 文档协作翻译小组，原文：Disclosed security issues in Django。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

点击劫持保护

点击劫持中间件和装饰器提供了简捷易用的，对点击劫持的保护。这种攻击在恶意站点诱导用户点击另一个站点的被覆盖元素时出现，另一个站点已经加载到了隐藏的 `frame` 或 `iframe` 中。

点击劫持的示例

假设一个在线商店拥有一个页面，已登录的用户可以点击“现在购买”来购买一个商品。用户为了方便，可以选择一直保持商店的登录状态。一个攻击者的站点可能在他们自己的页面上会创建一个“我喜欢Ponies”的按钮，并且在一个透明的 `iframe` 中加载商店的页面，把“现在购买”的按钮隐藏起来覆盖在“我喜欢Ponies”上。如果用户访问了攻击者的站点，点击“我喜欢Ponies”按钮会触发对“现在购买”按钮的无意识的点击，不知不觉中购买了商品。

点击劫持的防御

现代浏览器遵循X-Frame-Options协议头，它表明一个资源是否允许加载到 `frame` 或者 `iframe` 中。如果响应包含值为 `SAMEORIGIN` 的协议头，浏览器会在 `frame` 中只加载同源请求的的资源。如果协议头设置为 `DENY` ，浏览器会在加载 `frame` 时屏蔽所有资源，无论请求来自于哪个站点。

Django提供了一些简单的方法来在你站点的响应中包含这个协议头：

- 一个简单的中间件，在所有响应中设置协议头。
- 一系列的视图装饰器，可以用于覆盖中间件，或者只用于设置指定视图的协议头。

如何使用

为所有响应设置X-Frame-Options

要为你站点中所有的响应设置相同的 `X-Frame-Options` 值，将 `'django.middleware.clickjacking.XFrameOptionsMiddleware'` 设置为 `MIDDLEWARE_CLASSES` ：

```
MIDDLEWARE_CLASSES = (
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
)
```

这个中间件可以在startproject生成的设置文件中开启。

通常，这个中间件会为任何开放的 `HttpResponse` 设置 `X-Frame-Options` 协议头为 `SAMEORIGIN` 。如果你想用 `DENY` 来替代它，要设置 `X_FRAME_OPTIONS` ：

```
X_FRAME_OPTIONS = 'DENY'
```

使用这个中间件时可能会有一些视图，你并不想为它设置 `X-Frame-Options` 协议头。对于这些情况，你可以使用一个视图装饰器来告诉中间件不要设置协议头：

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_exempt

@xframe_options_exempt
def ok_to_load_in_a_frame(request):
    return HttpResponse("This page is safe to load in a frame on any site.")
```

为每个视图设置 X-Frame-Options

Django提供了以下装饰器来为每个基础视图设置 `X-Frame-Options` 协议头。

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_deny
from django.views.decorators.clickjacking import xframe_options_sameorigin

@xframe_options_deny
def view_one(request):
    return HttpResponse("I won't display in any frame!")

@xframe_options_sameorigin
def view_two(request):
    return HttpResponse("Display in a frame if it's from the same origin as me.")
```

注意你可以在中间件的连接中使用装饰器。使用装饰器来覆盖中间件。

限制

`X-Frame-Options` 协议头只在现代浏览器中保护点击劫持。老式的浏览器会忽视这个协议头，并且需要 其它点击劫持防范技巧 。

支持 X-Frame-Options 的浏览器

- Internet Explorer 8+
- Firefox 3.6.9+
- Opera 10.5+
- Safari 4+
- Chrome 4.1+

另见

浏览器对 `X-Frame-Options` 支持情况的完整列表。

> 译者：Django 文档协作翻译小组，原文：Clickjacking protection。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

跨站请求伪造保护

CSRF 中间件和模板标签提供对跨站请求伪造简单易用的防护。某些恶意网站上包含链接、表单按钮或者JavaScript ，它们会利用登录过的用户在浏览器中的认证信息试图在你的网站上完成某些操作，这就是跨站攻击。还有另外一种相关的攻击叫做“登录CSRF”，攻击站点触发用户浏览器用其它人的认证信息登录到其它站点。

防护CSRF 攻击的第一道防线是保证GET 请求（以及在9.1.1 Safe Methods, HTTP 1.1, **RFC 2616** 中定义的其它安全的方法）不会产生副作用。通过不安全的请求方法例如POST、PUT 和DELETE 可以通过以下步骤进行防护。

如何使用

想在你的视图中使用CSRF 防护，请遵循以下步骤：

1. CSRF 中间件在 `MIDDLEWARE_CLASSES` 设置中默认启用。如果你要覆盖这个设置，请记住 `'django.middleware.csrf.CsrfViewMiddleware'` 应该位于其它任何假设CSRF 已经处理过的视图中间件之前。

   如果你关闭了它，虽然不建议，你可以在你想要保护的视图上使用 `csrf_protect()` （见下文）。

2. 在使用POST 表单的模板中，对于内部的URL请在 `<form>` 元素中使用 `csrf_token` 标签：

   ```
   &lt;form action="." method="post"&gt;
   ```

   它不应该用于目标是外部URL 的POST 表单，因为这将引起CSRF 信息泄露而导致出现漏洞。

3. 在对应的视图函数中，确保使用 `'django.template.context_processors.csrf'` Context 处理器。通常可以用两种方法实现：

   i. 使用RequestContext，它会始终使用 `'django.template.context_processors.csrf'` （无论 `TEMPLATES` 设置中配置的是什么模板上下文处理器）。如果你正在使用通用视图或Contrib 中的应用，你就不用担心了，因为这些应用通篇都使用RequestContext。

   ii. 手工导入并使用处理器来生成CSRF token，并将它添加到模板上下文中。例如：

```
from django.shortcuts import render_to_response
from django.template.context_processors import csrf

def my_view(request):
    c = {}
    c.update(csrf(request))
    # ... view code here
    return render_to_response("a_template.html", c)
```

你可能想要编写你自己的 `render_to_response()` 来处理这个步骤。

AJAX

虽然上面的方法可以用于AJAX POST 请求，但是它不太方便：你必须记住在每个POST 请求的数据中传递CSRF token。由于这个原因，还有另外一种方法：在每个XMLHttpRequest 上设置一个自定义的 `X-CSRFToken` 头部，其值为CSRF token。这非常容易，因为许多JavaScript 框架都提供在每个请求上设置头部的方法。

第一步，你必须获得CSRF token。建议从 `csrftoken` Cookie 中获取，如果你在视图中启用CSRF 防护它就会设置。

注

CSRF token 的Cookie 默认叫做 `csrftoken`，你可以通过 `CSRF_COOKIE_NAME` 设置自定义它的名字。

获取token 非常简单：

```
// using jQuery
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie != '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // Does this cookie string begin with the name we wa
nt?
            if (cookie.substring(0, name.length + 1) == (name +
'=')) {
                cookieValue = decodeURIComponent(cookie.substrin
g(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}
var csrftoken = getCookie('csrftoken');
```

以上代码可以使用jQuery cookie plugin 来替换 `getCookie` ：

```
var csrftoken = $.cookie('csrftoken');
```

注

CSRF token 也存在于DOM 中，但只有你在模板中明确使用 `csrf_token` 标签时才有。Cookie 中包含标准的token； `CsrfViewMiddleware` 倾向于使用Cookie 而不是DOM 中的token。无论如何，如果DOM 中具有token，Cookie 中将保证会有，所以你应该使用Cookie。

警告

如果你的视图渲染的模板没有包含 `csrf_token` 标签，Django 可能不会再Cookie 中设置CSRF token。这常见于表单是动态的方式添加到网页中的。为了解决这个问题，Django 提供一个视图装饰器 `ensure_csrf_cookie()` ，它将强制设置这个Cookie。

最后，你不想在AJAX 请求中设置头部，使用jQuery 已经更新版本的settings.crossDomain 可以包含CSRF token 不会发送给其它域：

```
function csrfSafeMethod(method) {
    // these HTTP methods do not require CSRF protection
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type) && !this.crossDomain)
 {
            xhr.setRequestHeader("X-CSRFToken", csrftoken);
        }
    }
});
```

其它模版引擎

当使用非Django自带的模板引擎，你可以手动设置token，但要确保它在template context中可用

例如，在采用Jinja2模板引擎时，你可以在表单中包含以下内容：

```
<div style="display:none">
    <input type="hidden" name="csrfmiddlewaretoken" value="nK3w9
RcmP6lHX8mUKQxmFRL49osJDl5X">
</div>
```

您可以使用与上述*AJAX code*类似的JavaScript来获取CSRF令牌的值。

装饰器方法

您可以对需要保护的特定视图使用 `csrf_protect` 装饰器，而不是添加 `CsrfViewMiddleware` 作为整体保护。对于在输出中插入CSRF令牌的视图以及接受POST表单数据的视图，必须使用**both**。（这些通常是相同的视图函数，但不总是）。

使用装饰器本身是不推荐的，因为如果你忘记使用它，你会有一个安全漏洞。使用两者的“belt and braces”策略是很好的，并且将产生最小的开销。

`csrf_protect` (*view*)

装饰器为视图提供 `CsrfViewMiddleware` 的保护。

用法：

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

如果您使用的是基于类的视图，则可以参考装饰基于类的视图。

已拒绝的请求

默认情况下，如果传入请求未能通过 `CsrfViewMiddleware` 执行的检查，则向用户发送“403禁止”响应。这通常只有在有一个真正的跨站点请求伪造，或由于编程错误，CSRF令牌未包括在POST表单中的情况下才会看到。

错误页面，但是，不是很友好，所以你可能想提供自己的视图来处理这种情况。为此，只需设置 `CSRF_FAILURE_VIEW` 设置即可。

它是如何工作的

跨站伪造保护基于以下几点：

1. 设置为随机值（与session无关的随机数）的CSRF Cookie，其他网站将无法访问。

   此Cookie由 `CsrfViewMiddleware` 设置。它是永久的，但由于没有办法设置一个永远不会过期的cookie，它会与每个响应一起发送，该响应调用 `django.middleware.csrf.get_token()` （使用的函数内部检索CSRF令牌）。

2. 所有传出POST表单中都有一个名为“csrfmiddlewaretoken”的隐藏表单字段。此字段的值是CSRF cookie的值。

   此部分由模板标记完成。

3. 对于所有未使用HTTP GET，HEAD，OPTIONS或TRACE的传入请求，必须存在CSRF cookie，并且“csrfmiddlewaretoken”字段必须存在且正确。如果不是，用户将得到403错误。

   此检查由 `CsrfViewMiddleware` 完成。

4. 此外，对于HTTPS请求， `CsrfViewMiddleware` 会执行严格的引用程序检查。这是必要的，以解决在使用独立于会话的现时在HTTPS下是可能的中间人中间人攻击，由于HTTP'Set-Cookie'头被（不幸）被接受由与站点。（由于

> Referer头的存在在HTTP下不够可靠，因此不会对HTTP请求执行Referer检查。）

这确保只有源自您的网站的表单才能用于POST数据。

它故意忽略GET请求（以及由 **RFC 2616**定义为“安全”的其他请求）。这些请求不应该有任何潜在的危险副作用，因此具有GET请求的CSRF攻击应该是无害的。**RFC 2616**将POST，PUT和DELETE定义为“不安全”，并且假定所有其他方法都不安全，以获得最大保护。

Caching

如果模板使用了 `csrf_token` 模板标签（或者 `get_token` 函数被称为某种其他方式），则 `CsrfViewMiddleware` 将添加一个Cookie和 `Vary: Cookie` 标头。这意味着如果按照指示使用中间件（ `UpdateCacheMiddleware` 在所有其他中间件之前），则中间件将与高速缓存中间件良好匹配。

但是，如果在单个视图上使用缓存装饰器，CSRF中间件将无法设置Vary头或CSRF cookie，并且响应将被缓存而没有任何一个。在这种情况下，在任何需要插入CSRF令牌的视图中，您应该首先使用 `django.views.decorators.csrf.csrf_protect()` 装饰器：

```
from django.views.decorators.cache import cache_page
from django.views.decorators.csrf import csrf_protect

@cache_page(60 * 15)
@csrf_protect
def my_view(request):
    ...
```

如果您使用的是基于类的视图，则可以参考装饰基于类的视图。

测试

`CsrfViewMiddleware` 通常会阻碍测试视图函数，因为需要每次POST请求都必须发送CSRF令牌。出于这个原因，Django的HTTP客户端测试已被修改，以设置一个标志，请求放松中间件和 `csrf_protect` 装饰器，以便他们不再拒绝请求。在每个其他方面（例如发送cookie等），它们的行为相同。

如果由于某种原因，您希望测试客户端执行CSRF检查，您可以创建实施CSRF检查的测试客户端的实例：

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

限制

网站中的子网域可以在整个网域的客户端上设置Cookie。通过设置cookie并使用相应的令牌，子域将能够绕过CSRF保护。避免这种情况的唯一方法是确保子域由受信任的用户控制（或至少无法设置Cookie）。请注意，即使没有CSRF，也有其他漏洞，如会话固定，使得给不受信任的方的子域一个坏主意，这些漏洞不能轻易地用当前浏览器修复。

边框

某些视图可能有不寻常的要求，这意味着它们不适合这里设想的正常模式。在这些情况下，许多实用程序可能很有用。以下部分描述了可能需要的方案。

实用程序

下面的示例假设您使用基于函数的视图。如果您使用基于类的视图，则可以参考装饰基于类的视图。

`csrf_exempt` (*view*)[source]

这个装饰器将视图标记为不受中间件保护的保护。例：

```
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse

@csrf_exempt
def my_view(request):
    return HttpResponse('Hello world')
```

`requires_csrf_token` (*view*)

通常，如果 `CsrfViewMiddleware.process_view` 或类似 `csrf_protect` 的等效项未运行，则 `csrf_token` 模板标记将无法正常工作。视图装饰器 `requires_csrf_token` 可用于确保模板标记正常工作。此装饰器与 `csrf_protect` 类似，但不会拒绝传入的请求。

例：

```
from django.views.decorators.csrf import requires_csrf_token
from django.shortcuts import render

@requires_csrf_token
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

`ensure_csrf_cookie` (*view*)

此装饰器强制视图发送CSRF cookie。

场景

CSRF保护应该禁用只有几个视图

大多数视图需要CSRF保护，但有几个不需要。

解决方案：而不是禁用中间件，并对所有需要它的视图应用 `csrf_protect` ，启用中间件并使用 `csrf_exempt()` 。

CsrfViewMiddleware.process_view未使用

有些情况下，如果 `CsrfViewMiddleware.process_view` 可能在您的视图运行之前没有运行 - 例如404和500处理程序，但是您仍然需要一个表单中的CSRF令牌。

解决方案：使用 `requires_csrf_token()`

不受保护的视图需要CSRF令牌

可能有一些视图未受保护，并且已被 `csrf_exempt` 豁免，但仍需要包括CSRF令牌。

解决方案：使用 `csrf_exempt()` ，后跟 `requires_csrf_token()` 。（即 `requires_csrf_token` 应该是最内部的装饰器）。

View需要保护一个路径

视图仅需要一组条件下的CSRF保护，并且在其余时间内不能拥有它。

解决方案：对于需要保护的路径，使用 `csrf_exempt()` 作为整个视图函数， `csrf_protect()` 例：

```
from django.views.decorators.csrf import csrf_exempt, csrf_prote
ct

@csrf_exempt
def my_view(request):

    @csrf_protect
    def protected_path(request):
        do_something()

    if some_condition():
       return protected_path(request)
    else:
       do_something_else()
```

页面使用AJAX，没有任何HTML表单

网页通过ajax发出post请求，并且该网页没有带有 `csrf_token` 的HTML表单，就会导致所需的CSRF Cookie被发送。

解决方案：在发送页面的视图上使用 `ensure_csrf_cookie()` 。

Contrib和可重复使用的应用程序

因为开发人员可以关闭 `CsrfViewMiddleware` ，所以contrib应用程序中的所有相关视图都使用 `csrf_protect` 装饰器，以确保这些应用程序的安全性不受CSRF的影响。建议其他需要相同保证的可重用应用程序的开发人员也在其视图上使用 `csrf_protect` 装饰器。

设置

许多设置可用于控制Django的CSRF行为：

- CSRF_COOKIE_AGE
- CSRF_COOKIE_DOMAIN
- CSRF_COOKIE_HTTPONLY
- CSRF_COOKIE_NAME
- CSRF_COOKIE_PATH
- CSRF_COOKIE_SECURE
- CSRF_FAILURE_VIEW

加密签名

web应用安全的黄金法则是，永远不要相信来自不可信来源的数据。有时通过不可信的媒介来传递数据会非常方便。密码签名后的值可以通过不受信任的途径传递，这样是安全的，因为任何篡改都会检测的到。

Django提供了用于签名的底层API，以及用于设置和读取被签名cookie的上层API，它们是web应用中最常使用的签名工具之一。

你可能会发现，签名对于以下事情非常有用：

- 生成用于“重置我的账户”的URL，并发送给丢失密码的用户。
- 确保储存在隐藏表单字段的数据不被篡改，
- 生成一次性的秘密URL，用于暂时性允许访问受保护的资源，例如用户付费的下载文件。

保护 SECRET_KEY

当你使用 startproject创建新的Django项目时，自动生成的 settings.py文件会得到一个随机的SECRET_KEY值。这个值是保护签名数据的密钥 -- 它至关重要，你必须妥善保管，否则攻击者会使用它来生成自己的签名值。

使用底层 API

Django的签名方法存放于`django.core.signing`模块。首先创建一个 `Signer` 的实例来对一个值签名：

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

这个签名会附加到字符串末尾，跟在冒号后面。你可以使用`unsign` 方法来获取原始的值：

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

如果签名或者值以任何方式改变，会抛出`django.core.signing.BadSignature` 异常：

```
>>> from django.core import signing
>>> value += 'm'
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
```

通常，`Signer` 类使用 `SECRET_KEY` 设置来生成签名。你可以通过向 `Signer` 构造器传递一个不同的密钥来使用它：

```
>>> signer = Signer('my-other-secret')
>>> value = signer.sign('My string')
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

`class Signer(key=None, sep=':', salt=None)[source]`

返回一个 `signer` ，它使用 `key` 来生成签名，并且使用 `sep` 来分割值。 `sep` 不能是 [URL安全的base64字母表(http://tools.ietf.org/html/rfc4648#section-5)]中的字符。字母表含有数字、字母、连字符和下划线。

使用salt参数

如果你不希望对每个特定的字符串都生成一个相同的签名哈希值，你可以在 `Signer` 类中使用可选的 `salt` 参数。使用 `salt` 参数会同时用它和 `SECRET_KEY` 初始化签名哈希函数：

```
>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw')
'My string'
```

以这种方法使用 `salt` 会把不同的签名放在不同的命名空间中。来自于单一命名空间（一个特定的 `salt` 值）的签名不能用于在不同的命名空间中验证相同的纯文本字符串。不同的命名空间使用不同的 `salt` 设置。这是为了防止攻击者使用在一个地方的代码中生成的签名后的字符串，作为使用不同 `salt` 来生成（和验证）签名的另一处代码的输入。

不像你的 `SECRET_KEY` ，你的 `salt` 参数可以不用保密。

验证带有时间戳的值

`TimestampSigner` 是 `Signer` 的子类，它向值附加一个签名后的时间戳。这可以让你确认一个签名后的值是否在特定时间段之内被创建：

```
>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'
```

`class TimestampSigner(key=None, sep=':', salt=None)[source]`

`sign(value)[source]`

签名 `value` ，并且附加当前的时间戳。

`unsign(value, max_age=None)[source]`

检查 `value` 是否在少于 `max_age` 秒之前被签名，如果不是则抛出 `SignatureExpired` 异常。 `max_age` 参数接受一个整数或者 `datetime.timedelta` 对象。

> Changed in Django 1.8:
>
> 在此之前， `max_age`参数只接受整数。

保护复杂的数据结构

如果你希望保护一个列表、元组或字典，你可以使用签名模块的 `dumps` 和 `loads` 函数来实现。它们模仿了Python的 `pickle` 模块，但是在背后使用了 `JSON` 序列化。 `JSON` 可以确保即使你的 `SECRET_KEY` 被盗取，攻击者并不能利用 `pickle` 的格式来执行任意的命令：

```
>>> from django.core import signing
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1NMg1b:zGcDE4-TCkaeGzLeW9UQwZesciI'
>>> signing.loads(value)
{'foo': 'bar'}
```

由于 `JSON` 的本质（列表和元组之间没有原生的区别），如果你传进来一个元组，你会从 `signing.loads(object)` 得到一个列表：

```
>>> from django.core import signing
>>> value = signing.dumps(('a','b','c'))
>>> signing.loads(value)
['a', 'b', 'c']
```

`dumps(obj, key=None, salt='django.core.signing', compress=False)[sou`

返回URL安全，sha1签名的base64压缩的JSON字符串。序列化的对象使用 `TimestampSigner` 来签名。

`loads(string, key=None, salt='django.core.signing', max_age=None)[so`

`dumps()` 的反转，如果签名失败则抛出 `BadSignature` 异常。如果提供了 `max_age` 则会检查它（以秒为单位）。

> 译者：Django 文档协作翻译小组，原文：Cryptographic signing。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

中间件

这篇文档介绍了Django自带的所有中间件组件。 要查看关于如何使用它们以及如何编写自己的中间件，请见中间件使用指导。

可用的中间件

缓存中间件

class `UpdateCacheMiddleware` [source]

class `FetchFromCacheMiddleware` [source]

开启全站范围的缓存。 如果开启了这些缓存，任何一个由Django提供的页面将会被缓存，缓存时长是由你在 `CACHE_MIDDLEWARE_SECONDS` 配置中定义的。详见缓存文档。

“通用”的中间件

class `CommonMiddleware` [source]

给完美主义者增加一些便利条件：

- 禁止访问 `DISALLOWED_USER_AGENTS` 中设置的用户代理，这项配置应该是一个已编译的正则表达式对象的列表。
- 基于 `APPEND_SLASH` 和 `PREPEND_WWW` 的设置来重写URL。

  如果 `APPEND_SLASH` 设为 `True` ，并且初始URL 没有以斜线结尾以及在URLconf 中没找到对应定义，这时形成一个斜线结尾的新URL。如果这个新的URL存在于URLconf，那么Django 重定向请求到这个新URL上。否则，按正常情况处理初始的URL。

  比如，如果你没有为 `foo.com/bar` 定义有效的正则，但是为 `foo.com/bar/` 定义了有效的正则， `foo.com/bar` 将会被重定向到 `foo.com/bar/` 。

  如果 `PREPEND_WWW` 设为 `True` ，前面缺少 "www."的url将会被重定向到相同但是以一个"www."开头的url。

  两种选项都是为了规范化url。其中的哲学就是，任何一个url应该在一个地方仅存在一个。技术上来讲，URL `foo.com/bar` 区别于 `foo.com/bar/` —— 搜索引擎索引会把这里分开处理 —— 因此最佳实践就是规范化URL。

- 基于 `USE_ETAGS` 设置来处理ETag。如果设置 `USE_ETAGS` 为 `True` ，Django会通过MD5-hashing处理页面的内容来为每一个页面请求计算Etag，并且如果合适的话，它将会发送携带 `Not Modified` 的响应。

`CommonMiddleware.response_redirect_class`

New in Django 1.8.

默认为 `HttpResponsePermanentRedirect` 。它继承了 `CommonMiddleware` ，并覆写了属性来自定义中间件发出的重定向。

class `BrokenLinkEmailsMiddleware` [source]

- 向 `MANAGERS` 发送死链提醒邮件（详见错误报告）。

GZip中间件

class `GZipMiddleware` [source]

警告

安全研究员最近发现，当压缩技术（包括 `GZipMiddleware` ）用于一个网站的时候，网站会受到一些可能的攻击。此外，这些方法可以用于破坏Django的CSRF保护。在你的站点使用 `GZipMiddleware` 之前，你应该先仔细考虑一下你的站点是否容易受到这些攻击。 如果你不确定是否会受到这些影响，应该避免使用? `GZipMiddleware` 。详见the BREACH paper (PDF)和breachattack.com。

为支持GZip压缩的浏览器（一些现代的浏览器）压缩内容。

建议把这个中间件放到中间件配置列表的第一个，这样压缩响应内容的处理会到最后才 发生。

如果满足下面条件的话，内容不会被压缩：

- 消息体的长度小于200个字节。
- 响应已经设置了 `Content-Encoding` 协议头。
- 请求（浏览器）没有发送包含 `gzip` 的 `Accept-Encoding` 协议头。

你可以通过这个 `gzip_page()` 装饰器使用独立的GZip压缩。

带条件判断的GET中间件

class `ConditionalGetMiddleware` [source]

处理带有条件判断状态GET操作。 如果一个请求包含 `ETag` ?或者 `Last-Modified` 协议头，并且请求包含 `If-None-Match` 或 `If-Modified-Since` ，这时响应会被 替换为 `HttpResponseNotModified` 。

另外，它会设置 `Date` 和 `Content-Length` 响应头。

地域性中间件

class `LocaleMiddleware` [source]

基于请求中的数据开启语言选择。 它可以为每个用户进行定制。 详见国际化文档。

`LocaleMiddleware.response_redirect_class`

默认为 `HttpResponseRedirect` 。继承自 `LocaleMiddleware` 并覆写了属性来自定义中间件发出的重定向。

消息中间件

class `MessageMiddleware` [source]

开启基于Cookie和会话的消息支持。详见消息文档。

安全中间件

警告

如果你的部署环境允许的话，让你的前端web服务器展示 `SecurityMiddleware` 提供的功能是个好主意。这样一来，如果有任何请求没有被Django处理（比如静态媒体或用户上传的文件），它们会拥有和向Django 应用的请求相同的保护。

class `SecurityMiddleware` [source]

New in Django 1.8.

`django.middleware.security.SecurityMiddleware` 为请求/响应循环提供了几种安全改进。每一种可以通过一个选项独立开启或关闭。

- SECURE_BROWSER_XSS_FILTER
- SECURE_CONTENT_TYPE_NOSNIFF
- SECURE_HSTS_INCLUDE_SUBDOMAINS
- SECURE_HSTS_SECONDS
- SECURE_REDIRECT_EXEMPT
- SECURE_SSL_HOST
- SECURE_SSL_REDIRECT

HTTP Strict Transport Security (HSTS)

对于那些应该只能通过HTTPS访问的站点，你可以通过设置HSTS协议头，通知现代的浏览器，拒绝用不安全的连接来连接你的域名。这会降低你受到SSL-stripping的中间人（MITM）攻击的风险。

如果你将 `SECURE_HSTS_SECONDS` 设置为一个非零值， `SecurityMiddleware` 会在所有的HTTPS响应中设置这个协议头。

开启HSTS的时候，首先使用一个小的值来测试它是个好主意，例如，让 `SECURE_HSTS_SECONDS = 3600` 为一个小时。每当浏览器在你的站点看到HSTS协议头，都会在提供的时间段内拒绝使用不安全（HTTP）的方式连接到你的域名。一旦你确认你站点上的所有东西都以安全的方式提供（例如，HSTS并不会干扰任何事情），建议你增加这个值，这样不常访问你站点的游客也会被保护（比如，一般设置为31536000秒，一年）。

另外，如果你将? `SECURE_HSTS_INCLUDE_SUBDOMAINS` 设置为 `True` ,， `SecurityMiddleware` 会将 `includeSubDomains` 标签添加到 `Strict-Transport-Security` 协议头中。强烈推荐这样做（假设所有子域完全使用HTTPS），否则你的站点仍旧有可能由于子域的不安全连接而受到攻击。

警告

HSTS策略在你的整个域中都被应用，不仅仅是你所设置协议头的响应中的url。所以，如果你的整个域都设置为HTTPS only，你应该只使用HSTS策略。

适当遵循HSTS协议头的浏览器，会通过显示警告的方式，拒绝让用户连接到证书过期的、自行签署的、或者其他SSL证书无效的站点。如果你使用了HSTS，确保你的证书处于一直有效的状态！

注意

如果你的站点部署在负载均衡器或者反向代理之后，并且 `Strict-Transport-Security` 协议头没有添加到你的响应中，原因是Django有可能意识不到这是一个安全连接。你可能需要设置 `SECURE_PROXY_SSL_HEADER` 。

X-Content-Type-Options: nosniff

一些浏览器会尝试猜测他们所得内容的类型，而不是读取 `Content-Type` 协议头。虽然这样有助于配置不当的服务器正常显示内容，但也会导致安全问题。

如果你的站点允许用户上传文件，一些恶意的用户可能会上传一个精心构造的文件，当你觉得它无害的时候，文件会被浏览器解释成HTML或者Javascript。

欲知更多有关这个协议头和浏览器如何处理它的内容，你可以在IE安全博客中读到它。

要防止浏览器猜测内容类型，并且强制它一直使用? `Content-Type` 协议头中提供的类型，你可以传递 `X-Content-Type-Options: nosniff` 协议头。 `SecurityMiddleware` 将会对所有响应这样做，如果 `SECURE_CONTENT_TYPE_NOSNIFF` 设置为 `True` 。

注意在大多数Django不涉及处理上传文件的部署环境中，这个设置不会有任何帮助。例如，如果你的 `MEDIA_URL` 被前端web服务器直接处理（例如nginx和Apache），你可能想要在那里设置这个协议头。而在另一方面，如果你使用

Django执行为了下载文件而请求授权之类的事情，并且你不能使用你的web服务器设置协议头，这个设置会很有用。

X-XSS-Protection: 1; mode=block

一些浏览器能够屏蔽掉出现XSS攻击的内容。通过寻找页面中GET或者POST参数中的JavaScript内容来实现。如果JavaScript在服务器的响应中被重放，页面就会停止渲染，并展示一个错误页来取代。

X-XSS-Protection协议头用来控制XSS过滤器的操作。

要在浏览器中启用XSS过滤器，并且强制它一直屏蔽可疑的XSS攻击，你可以在协议头中传递 `X-XSS-Protection: 1; mode=block` 。如果 `SECURE_BROWSER_XSS_FILTER` 设置为 `True` ， `SecurityMiddleware` 会在所有响应中这样做。

警告

浏览器的XSS过滤器是一个十分有效的手段，但是不要过度依赖它。它并不能检测到所有的XSS攻击，也不是所有浏览器都支持这一协议头。确保你校验和过滤了所有的输入来防止XSS攻击。

SSL重定向

如果你同时提供HTTP和HTTPS连接，大多数用户会默认使用不安全的（HTTP）链接。为了更高的安全性，你应该重定向所有的HTTP 连接到HTTPS。

如果你将 `SECURE_SSL_REDIRECT` 设置为True， `SecurityMiddleware` 会将HTTP链接永久地（HTTP 301，permanently）重定向到HTTPS连接。

注意

由于性能因素，最好在Django外面执行这些重定向，在nginx这种前端负载均衡器或者反向代理服务器中执行。 `SECURE_SSL_REDIRECT` 专门为这种部署情况而设计，当这不可选择的时候。

如果 `SECURE_SSL_HOST` 设置有一个值，所有重定向都会发到值中的主机，而不是原始的请求主机。

如果你站点上的一些页面应该以HTTP方式提供，并且不需要重定向到HTTPS，你可以 `SECURE_REDIRECT_EXEMPT` 设置中列出匹配那些url的正则表达式。

注意

如果你在负载均衡器或者反向代理服务器后面部署应用，而且Django不能辨别出什么时候一个请求是安全的，你可能需要设置 `SECURE_PROXY_SSL_HEADER` 。

会话中间件

class `SessionMiddleware` [source]

开启会话支持。详见会话文档。

站点中间件

class `CurrentSiteMiddleware` [source]

New in Django 1.7.

向每个接收到的 `HttpRequest` 对象添加一个 `site` 属性，表示当前的站点。详见站点文档。

认证中间件

class `AuthenticationMiddleware` [source]

向每个接收到的 `HttpRequest` 对象添加 `user` 属性，表示当前登录的用户。详见*web*请求中的认证。

class `RemoteUserMiddleware` [source]

使用web服务器提供认证的中间件。详见使用*REMOTE_USER*进行认证。

class `SessionAuthenticationMiddleware` [source]

New in Django 1.7.

当用户修改密码的时候使用户的会话失效。详见密码更改时的会话失效。在 `MIDDLEWARE_CLASSES` 中，这个中间件必须出现在 `django.contrib.auth.middleware.AuthenticationMiddleware` 之后。

CSRF保护中间件

class `CsrfViewMiddleware` [source]

添加跨站点请求伪造的保护，通过向POST表单添加一个隐藏的表单字段，并检查请求中是否有正确的值。详见*CSRF*保护文档。

X-Frame-Options中间件

class `XFrameOptionsMiddleware` [source]

通过*X-Frame-Options*协议头进行简单的点击劫持保护。

中间件的排序

下面是一些关于Django中间件排序的提示。

1. `UpdateCacheMiddleware`

   放在修改 `大量` 协议头的中间件(`SessionMiddleware` , `GZipMiddleware` , `LocaleMiddleware`)之前。

2. `GZipMiddleware`

   放在任何可能修改或使用响应消息体的中间件之前。

   放在 `UpdateCacheMiddleware` 之后：会修改 `大量` 的协议头。

3. `ConditionalGetMiddleware`

   放在 `CommonMiddleware` 之前：当 `USE_ETAGS` = `True` 时会使用它的 `Etag` 协议头。

4. `SessionMiddleware`

   放在 `UpdateCacheMiddleware` 之后：会修改 `大量` 协议头。

5. `LocaleMiddleware`

   放在 `SessionMiddleware` （由于使用会话数据）和? `CacheMiddleware` （由于要修改 `大量` 协议头）之后的最上面。

6. `CommonMiddleware`

   放在任何可能修改相应的中间件之前（因为它会生成 `ETags` ）。

   在 `GZipMiddleware` 之后，不会在压缩后的内容上再去生成 `ETag` 。

   尽可能放在靠上面的位置，因为 `APPEND_SLASH` 或者 `PREPEND_WWW` 设置为? `True` 时会被重定向。

7. `CsrfViewMiddleware`

   放在任何假设CSRF攻击被处理的视图中间件之前。

8. `AuthenticationMiddleware`

   放在 `SessionMiddleware` 之后：因为它使用会话存储。

9. `MessageMiddleware`

   放在 `SessionMiddleware` 之后：会使用基于会话的存储。

10. `FetchFromCacheMiddleware`

    放在任何修改 `大量` 协议头的中间件之后：协议头被用来从缓存的哈希表中获取值。

11. `FlatpageFallbackMiddleware`

    应该放在最底下，因为它是中间件中的最后一手。

12. `RedirectFallbackMiddleware`

    应该放在最底下，因为它是中间件中的最后一手。

国际化和本地化

Django 提供了一种健壮的国际化和本地化框架来帮助你实现多种语言和世界区域范围的开发。

国际化和本地化

概述

国际化和本地化的目的就是让一个网站应用能做到根据用户语种和指定格式的不同而提供不同的内容。

Django 对文本翻译, 日期、时间和数字的格式化，以及时区提供了完善的支持。

实际上，Django做了两件事：

- 由开发者和模板作者指定应用的哪些部分应该翻译，或是根据本地语种和文化进行相应的格式化。
- 根据用户的偏好设置，使用钩子将web应用本地化。

很显然，翻译取决于用户所选语言，而格式化通常取决于用户所在国家。 这些信息由浏览器通过 `Accept-Language` 协议头提供。不过确定时区就不是这么简单了。

定义

国际化和本地化通常会被混淆，这里我们对其进行简单的定义和区分：

国际化

让软件支持本地化的准备工作，通常由开发者完成。

本地化

编写翻译和本地格式，通常由翻译者完成。

更多细节详见W3C Web Internationalization FAQ、Wikipedia article和GNU gettext documentation。

> 警告
>
> 是否启用翻译和格式化分别由配置项 `USE_I18N` 和 `USE_L10N` 决定。 但是，这两个配置项都同时影响国际化和本地化。 这种情况是Django的历史因素所致。

下面几项可帮助我们更好地处理某种语言：

本地化名称

表示地域文化的名称，可以是 `ll` 格式的语种代码，也可以是 `ll_CC` 格式的语种和国家组合代码。例如： `it` , `de_AT` , `es` , `pt_BR` 。语种部分总是小写而国家部分则应是大写，中间以下划线(_)连接。

语言代码

表示语言的名称。浏览器会发送带有语言代码的 `Accept-Language` HTTP报头给服务器。例如： `it` ， `de-at` ， `es` ， `pt-br` . 语种和国家部分都是小写，中间以破折线(-)连接。

消息文件

消息文件是纯文本文件，包含某种语言下所有可用的翻译字符串及其对应的翻译结果。消息文件以 `.po` 做为文件扩展名。

翻译字符串

可以被翻译的文字。

格式文件

针对某个地域定义数据格式的Python模块。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

翻译

概述

为了让Django项目可翻译，你必须添加一些钩子到你的Python 代码和模板中。这些钩子叫做翻译字符串。它们告诉Django："如果这个文本的翻译可用，应该将它翻译成终端用户的语言。"你需要标记这些可翻译的字符串；系统只会翻译它知道的字符串。

Django 提供一些工具用于提取翻译字符串到消息文件中。这个文件方便翻译人员提供翻译字符串的目标语言。翻译人员填充完消息文件后，必须编译它。这个过程依赖GNU gettext 工具集。

完成这些事情之后，Django 将负责根据用户的语言偏好将网页翻译成对应的语言。

Django 的国际化钩子默认是打开的，这表示框架的某些地方已经有相关的I18N 钩子。如果你不需要使用国际化，你可以花两秒钟将在设置文件中设置 `USE_I18N = False` 。 这样的话，Django 将做一些优化而不加载国际化的机制。

注

还有一个独立但是相关的设置 `USE_L10N` ，它控制Django 是否应该实现本地化格式。更多信息参见本地化格式。

注

确保你的项目已经启用翻译（最快的方法是检查 `MIDDLEWARE_CLASSES` 是否包含 `django.middleware.locale.LocaleMiddleware` ）。如果还没有，请参见*Django 如何发现语言偏好*。

国际化：在**Python** 代码中

标准的翻译

函数 `ugettext()` 用于指定标准的翻译。习惯上会将它导入成一个别名 `_` 以节省打字。

注

Python 的标准库 `gettext` 模块将 `_()` 安装进全局命名空间中，并作为 `gettext()` 的别名。在Django 中，我们选择不遵守这个实践，原因有：

1. 对国际化的字符集（Unicode）的支持， `ugettext()` 比 `gettext()` 更有用。有时候，对于特定的文件，你应该使用 `ugettext_lazy()` 作为默认的翻

译方法。如果 `_()` 不在全局命名空间中，开发人员必须想清楚哪一个是最合适的翻译函数。
2. 下划线字符（ `_` ）在Python 的交互式shell 和doctest 测试中，用于表示“前一个结果”。安装全局的 `_()` 函数会引起混乱。显式地导入 `ugettext()` 为 `_()` 将避免这个问题。

在下面的示例中，文本 `"Welcome to my site."` 标记为一个翻译字符串：

```
from django.utils.translation import ugettext as _
from django.http import HttpResponse

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

很明显，你可以不用别名来编写这段代码。下面的例子与前面完全一样：

```
from django.utils.translation import ugettext
from django.http import HttpResponse

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

翻译在计算生成的值上进行。下面的例子与前面两个完全一样：

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

翻译可以在变量上进行。下面同样是一个完全一样的示例：

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(告诫使用变量或计算的值，如在前面的两个例子， 是Django的翻译字符串检测工具， `django-admin makemessages` ，将不能够找到这些字符串。 后面有 `makemessages` 的更多信息）。

传递给 `_()` 或 `ugettext()` 的字符串可以通过Python 标准的命名字符串插值语法接收占位符。示例：

```
def my_view(request, m, d):
    output = _('Today is %(month)s  %(day)s.') % {'month': m, 'd
ay': d}
    return HttpResponse(output)
```

这种技术可以让语言相关的翻译重新排序。例如，英语翻译可能是 `"Today is November 26."` ，而西班牙语可能是 `"Hoy es 26 de Noviembre."` —— 月份和天数的占位符交换位置了。

由于这个原因，每当有多个参数的时候，你都应该使用命名的字符串插值（例如， `%(day)s` ）而不是位置插值（例如， `%s` 或 `%d` ）。如果使用位置插值，翻译将不能重新排序占位符。

给翻译人员的注释

如果你想给翻译人员一些提示，可以添加一个以 `Translators` 为前缀的注释，例如：

```
def my_view(request):
    # Translators: This message appears on the home page only
    output = ugettext("Welcome to my site.")
```

这个注释会在生成的 `.po` 文件中出现在可翻译的结构上方，而且可以在大部分翻译工具中显示出来。

注

只是为了完整性，下面是生成的 `.po` 文件片段：

```
#. Translators: This message appears on the home page only
# path/to/python/file.py:123
msgid "Welcome to my site."
msgstr ""
```

它在模板中也可以工作。更多细节参见模板中给翻译人员的注释。

标记字符串为**no-op**

`django.utils.translation.ugettext_noop()` 函数用于标记字符串为一个翻译字符串但是不用翻译它。该字符串将在后面依据一个变量翻译。

如果你的常量字符串需要在不同的系统和用户之间交互 —— 例如数据库中的字符串，它们应该保存在源语言中，但是需要在最后例如呈现给用户的时刻翻译，可以使用它。

多元化

函数 `django.utils.translation.ungettext()` 用于指定多元化的消息。

`ungettext` 接收三个参数：单数形式的翻译字符串、复数形式的翻译字符串和对象的个数。

这个函数用于当你的Django 应用所要本地化的语言中，复数形式 比英语中的要复杂时（‘object’ 表示单数，‘objects’ 表示所有 `count` 不等于一的情形，无论具体的值是多少）。

例如：

```
from django.utils.translation import ungettext
from django.http import HttpResponse

def hello_world(request, count):
    page = ungettext(
        'there is %(count)d object',
        'there are %(count)d objects',
    count) % {
        'count': count,
    }
    return HttpResponse(page)
```

这个例子中对象的数字作为 `count` 变量传递给翻译语言

需要注意的是多元化是复杂的，在每种语言的工作方式不同。 `count` 计数到1的比较并不总是正确的规则。此代码看起来复杂，但会产生某些语言不正确的结果：

```
from django.utils.translation import ungettext
from myapp.models import Report

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural

text = ungettext(
    'There is %(count)d  %(name)s available.',
    'There are %(count)d  %(name)s available.',
    count
) % {
    'count': count,
    'name': name
}
```

不要尝试自己去实现单复数逻辑, 这样会出错。这种情况下, 可以考虑这么做:

```
text = ungettext(
    'There is %(count)d  %(name)s object available.',
    'There are %(count)d  %(name)s objects available.',
    count
) % {
    'count': count,
    'name': Report._meta.verbose_name,
}
```

注意

在使用 ungettext() 的时候,确保你你使用在每一个占位符中使用同一个名称。在上面的例子中,你可以注意到我们是怎么在两种翻译中使用 name 这个变量的。下面这个例子中,除了在某些语言中会表达不正确之外,还可能会造成错误:

```
text = ungettext(
    'There is %(count)d  %(name)s available.',
    'There are %(count)d  %(plural_name)s available.',
    count
) % {
    'count': Report.objects.count(),
    'name': Report._meta.verbose_name,
    'plural_name': Report._meta.verbose_name_plural
}
```

运行 django-admin compilemessages 时，你会得到一个错误:

```
a format specification for argument 'name', as in 'msgstr[0]', d
oesn't exist in 'msgid'
```

注意

复数形式和po文件

Django不支持在po文件中的自定义复数方程。当所有翻译目录被合并时，仅考虑主Django po文件的复数形式（在 django/conf/locale/<lang_code>/LC_MESSAGES/django.po ）。所有其他po文件中的多个表单将被忽略。因此，您不应在项目或应用程序po文件中使用不同的复数方程。

上下文标记

有时候，词语有几种含义，例如英语中的 "May" ，指的是月份名称和动词。要使翻译者能够在不同的上下文中正确翻译这些单词，您可以使用 django.utils.translation.pgettext() 函数或 django.utils.translation.npgettext()

在所得到的 `.po` 文件中，字符串将随着同一字符串存在不同的上下文标记而频繁出现（上下文将出现在 `msgctxt` 行）翻译给他们每个不同的翻译。

例如：

```
from django.utils.translation import pgettext

month = pgettext("month name", "May")
```

要么：

```
from django.db import models
from django.utils.translation import pgettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=pgettext_lazy(
        'help text for MyThing model', 'This is the help text'))
```

将出现在 `.po` 文件中：

```
msgctxt "month name"
msgid "May"
msgstr ""
```

上下文标记也受 `trans` 和 `blocktrans` 模板标记支持。

延迟翻译

使用 `django.utils.translation` 中的翻译函数的惰性版本（可以通过名称中的 `lazy` 后缀轻松识别）来平移字符串 - 当访问该值而不是当他们被叫。

这些函数存储对字符串的惰性引用 - 而不是实际的翻译。当字符串在字符串上下文中使用时，例如在模板呈现中，翻译本身将被完成。

当对这些函数的调用位于在模块加载时执行的代码路径中时，这是至关重要的。

这在定义模型，表单和模型表单时很容易发生，因为Django实现了这些，使得它们的字段实际上是类级别的属性。因此，请确保在以下情况下使用延迟翻译：

模型字段和关系 `verbose_name`

例如，要翻译以下模型中名称字段的帮助文本，请执行以下操作：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text')
)
```

您可以使用 `verbose_name` 选项将 `ForeignKey` ， `ManyToManyField` 或 `OneToOneField` 关系标记为可翻译的名称：

```
class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind, related_name='kinds',
                             verbose_name=_('kind'))
```

就像你在 `verbose_name` 中所做的那样，你应该为关系提供一个小写的详细名称文本，因为Django会在需要时自动定义它。

模型详细名称值

建议始终提供显式的 `verbose_name` 和 `verbose_name_plural` 选项，而不是依赖于以英语为中心的回退和有些朴素的确定django通过查看模型的类名执行的详细名称：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the
help text'))

    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('my things')
```

模型方法 `short_description`

对于模型方法，您可以使用 `short_description` 属性向Django和管理网站提供翻译：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind, related_name='kinds',
                             verbose_name=_('kind'))

    def is_mouse(self):
        return self.kind.type == MOUSE_TYPE
    is_mouse.short_description = _('Is it a mouse?')
```

使用延迟翻译对象

可以在Python中使用unicode字符串（类型为 `unicode` 的对象）的任何地方使用 `ugettext_lazy()` 调用的结果。如果您尝试在预期的字节（ `str` 对象）使用它，事情将无法正常工作，因为 `ugettext_lazy()` 对象不知道如何将自身转换为字节。你不能在一个bytestring中使用一个unicode字符串，所以这是正常的Python行为。例如：

```
# This is fine: putting a unicode proxy into a unicode string.
"Hello %s" % ugettext_lazy("people")

# This will not work, since you cannot insert a unicode object
# into a bytestring (nor can you insert our unicode proxy there)
b"Hello %s" % ugettext_lazy("people")
```

如果您看到像 `“hello &lt； django.utils.functional ...&gt；”` 的输出，将 `ugettext_lazy()` 的结果转换为字节。这是你的代码中的一个错误。

如果你不喜欢长的 `ugettext_lazy` 名称，可以将其命名为 `_` （下划线），如下所示：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text')
)
```

使用 `ugettext_lazy()` 和 `ungettext_lazy()` 在模型和效用函数中标记字符串是一种常见的操作。当你在代码中的其他地方使用这些对象时，你应该确保你不会意外地将它们转换为字符串，因为它们应该尽可能晚地转换（以便正确的区域设置生效）。这需要使用下面描述的辅助函数。

懒惰翻译和复数

当对多个字符串（ `[u]n[p]gettext_lazy` ）使用延迟转换时，通常不知道字符串定义时的 `number` 参数。因此，您有权将 `number` 参数传递一个键名称，而不是整数。然后在字符串插值期间，在该键下的字典中查找 `number` 。这里的例子：

```
from django import forms
from django.utils.translation import ungettext_lazy

class MyForm(forms.Form):
    error_message = ungettext_lazy("You only provided %(num)d argument",
        "You only provided %(num)d arguments", 'num')

    def clean(self):
        # ...
        if error:
            raise forms.ValidationError(self.error_message % {'num': number})
```

如果字符串只包含一个未命名的占位符，则可以直接使用 `number` 参数进行插值：

```
class MyForm(forms.Form):
    error_message = ungettext_lazy("You provided %d argument",
        "You provided %d arguments")

    def clean(self):
        # ...
        if error:
            raise forms.ValidationError(self.error_message % number)
```

连接字符串：string_concat()

标准的Python字符串连接（ `''.join([...])` ）不能用于包含延迟翻译对象的列表。相反，您可以使用 `django.utils.translation.string_concat()` ，创建一个延迟对象，连接它的内容和仅当结果包含在一个字符串。例如：

```
from django.utils.translation import string_concat
from django.utils.translation import ugettext_lazy
...
name = ugettext_lazy('John Lennon')
instrument = ugettext_lazy('guitar')
result = string_concat(name, ': ', instrument)
```

In this case, the lazy translations in `result` will only be converted to strings when `result` itself is used in a string (usually at template rendering time).

延迟翻译的其他用途

对于任何其他情况下，你想延迟翻译，但必须将可翻译字符串作为参数传递给另一个函数，你可以将这个函数包装在一个懒惰的调用自己。例如：

```
from django.utils import six  # Python 3 compatibility
from django.utils.functional import lazy
from django.utils.safestring import mark_safe
from django.utils.translation import ugettext_lazy as _

mark_safe_lazy = lazy(mark_safe, six.text_type)
```

然后稍后：

```
lazy_string = mark_safe_lazy(_("<p>My <strong>string!</strong></
p>"))
```

语言的本地化名称

`get_language_info` ()[source]

`get_language_info()` 函数提供有关语言的详细信息：

```
>>> from django.utils.translation import get_language_info
>>> li = get_language_info('de')
>>> print(li['name'], li['name_local'], li['bidi'])
German Deutsch False
```

字典的 `name` 和 `name_local` 属性分别包含英语和语言本身的语言名称。 `bidi` 属性仅对双向语言为True。

语言信息的来源是 `django.conf.locale` 模块。类似的访问此信息可用于模板代码。见下文。

国际化：在模板代码中

Django 模板 中的翻译使用两个模板标签，语法与Python 代码中使用的语法有稍许不同。为了让你的模板能够访问这些标签，需要将 `{% load i18n %}` 放置在模板的顶部。和所有模板标签一样，这个标签需要在所有使用翻译的模板中加载，即使这些模板扩展自已经加载 `i18n` 标签的模板。

反式 **template tag**

`{% trans %}` 模板标签翻译一个常量字符串（位于单引号或双引号中）或变量：

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

如果带有 `noop` 选项，变量的查找仍然继续但是会忽略翻译。这对于需要在未来进行翻译的内容非常有用：

```
<title>{% trans "myvar" noop %}</title>
```

在内部，内联的翻译使用 `ugettext()` 调用。

当一个模板变量（上面的 `myvar` ）传递给该标签时，该标签会首先在运行时刻将变量解析成一个字符串，然后在消息目录中查找这个字符串。

`{% trans %}` 不可以将模板标签嵌入到字符串中。如果你的翻译字符串需要带有变量（占位符），可以使用 `{% blocktrans %}` 。

如果你想提前翻译字符串但是不显示出来，你可以使用以下语法：

```
{% trans "This is the title" as the_title %}

<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

实际应用中，你将使用它来获取字符串，然后在多处使用或者作为参数传递给其它模板标签或过滤器：

```
{% trans "starting point" as start %}
{% trans "end point" as end %}
{% trans "La Grande Boucle" as race %}

<h1>
  <a href="/" title="{% blocktrans %}Back to '{{ race }}' homepa
ge{% endblocktrans %}">{{ race }}</a>
</h1>
<p>
{% for stage in tour_stages %}
    {% cycle start end %}: {{ stage }}{% if forloop.counter|divi
sibleby:2 %}<br />{% else %}, {% endif %}
{% endfor %}
</p>
```

利用 `context` 关键字，`{% trans %}` 还支持*contextual markers*：

```
{% trans "May" context "month name" %}
```

blocktrans template tag

与 `trans` 标签相反， `blocktrans` 标签允许您通过使用占位符来标记由文字和可变内容组成的复杂句子进行翻译：

```
{% blocktrans %}This string will have {{ value }} inside.{% endb
locktrans %}
```

要翻译模板表达式，例如访问对象属性或使用模板过滤器，您需要将表达式绑定到本地变量，以在翻译块中使用。例子：

```
{% blocktrans with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktrans %}

{% blocktrans with myvar=value|filter %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

您可以在单个 `blocktrans` 标记中使用多个表达式：

```
{% blocktrans with book_t=book|title author_t=author|title %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

注意

The previous more verbose format is still supported:
`{% blocktrans with book|title as book_t and author|title as author_t`

Other block tags (for example `{% for %}` or `{% if %}`) are not allowed inside a `blocktrans` tag.

如果解析其中一个块参数失败，则通过使用 `deactivate_all()` 函数暂时停用当前活动的语言，blocktrans将会回退到默认语言。

该标签还提供了复数。使用它：

- 指定并绑定名称为 `count` 的计数器值。该值将用于选择正确的复数形式。
- Specify both the singular and plural forms separating them with the `{% plural %}` tag within the `{% blocktrans %}` and `{% endblocktrans %}` tags.

一个例子：

```
{% blocktrans count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

一个更复杂的例子：

```
{% blocktrans with amount=article.price count years=i.length %}
That will cost $ {{ amount }} per year.
{% plural %}
That will cost $ {{ amount }} per {{ years }} years.
{% endblocktrans %}
```

除了计数器值之外，当您同时使用复数特性和绑定值到局部变量时，请记住 `blocktrans` 结构在内部转换为 `ungettext` 调用。这意味着相同的*notes regarding ungettext variables*适用。

反向URL查找不能在 `blocktrans` 中执行，应该预先检索（并存储）：

```
{% url 'path.to.view' arg arg2 as the_url %}
{% blocktrans %}
This is a URL: {{ the_url }}
{% endblocktrans %}
```

`{% blocktrans %}` also supports *contextual markers* using the `context` keyword:

```
{% blocktrans with name=user.username context "greeting" %}Hi {{
 name }}{% endblocktrans %}
```

`{% blocktrans %}` 支持的另一个功能是 `trimmed` 选项。此选项将从 `{% blocktrans %}` 的内容的开头和结尾删除换行符。标签，在行的开头和结尾替换任何空格，并使用空格字符将所有行合并为一个，以将它们分隔开。这对缩进 `{% blocktrans %}` 标记而不缩进缩进字符的内容非常有用在PO文件中的相应条目中，这使得翻译过程更容易。

例如，以下 `{% blocktrans %}` 标记：

```
{% blocktrans trimmed %}
  First sentence.
  Second paragraph.
{% endblocktrans %}
```

将导致输入 `“第一 句。 第二 PO文件，` `“ 第一 句。 第二` > `，如果未指定` trimmed `选项。`

Changed in Django 1.7:

已添加 `trimmed` 选项。

传递给标记和过滤器的字符串文字

您可以使用熟悉的 `_()` 语法将作为参数传递的字符串文字转换为标记和过滤器：

```
{% some_tag _("Page not found") value|yesno:_("yes,no") %}
```

在这种情况下，标记和过滤器都将看到翻译的字符串，因此他们不需要知道翻译。

注意

在此示例中，转换基础结构将传递字符串 `"yes,no"` ，而不是单个字符串 `"yes"` 和 `"no"` 翻译后的字符串需要包含逗号，以便过滤器解析代码知道如何拆分参数。例如，德语翻译者可以将字符串 `"yes,no"` 翻译为 `"ja,nein"` （保持逗号不变）。

模板中翻译员的评论

与*Python code*一样，可以使用注释指定这些笔记的注释，可以使用 `comment` 标签：

```
{% comment %}Translators: View verb{% endcomment %}
{% trans "View" %}

{% comment %}Translators: Short intro blurb{% endcomment %}
<p>{% blocktrans %}A multiline translatable
literal.{% endblocktrans %}</p>
```

或使用 `{#` ... `#}` *one-line comment constructs*：

```
<button type="submit">{% trans "Go" %}</button>

{% blocktrans %}Ambiguous translatable block of text{% endblocktrans %}
```

注意

为了完整性，这些是所得到的 `.po` 文件的相应片段：

```
#. Translators: View verb
# path/to/template/file.html:10
msgid "View"
msgstr ""

#. Translators: Short intro blurb
# path/to/template/file.html:13
msgid ""
"A multiline translatable"
"literal."
msgstr ""

# ...

#. Translators: Label of a button that triggers search
# path/to/template/file.html:100
msgid "Go"
msgstr ""

#. Translators: This is a text of the base template
# path/to/template/file.html:103
msgid "Ambiguous translatable block of text"
msgstr ""
```

在模板中切换语言

如果您要在模板中选择语言，可以使用 `language` 模板标记：

```
{% load i18n %}

{% get_current_language as LANGUAGE_CODE %}
<!-- Current language: {{ LANGUAGE_CODE }} -->
<p>{% trans "Welcome to our page" %}</p>

{% language 'en' %}
    {% get_current_language as LANGUAGE_CODE %}
    <!-- Current language: {{ LANGUAGE_CODE }} -->
    <p>{% trans "Welcome to our page" %}</p>
{% endlanguage %}
```

虽然第一次出现的“欢迎来到我们的页面”使用当前语言，第二次将始终是英语。

其他标签

这些标签还需要`{% 加载 i18n %}`。

- `{% get_available_languages 为 LANGUAGES %} 第一个元素是[_language`
- `{% get_current_language 作为 LANGUAGE_CODE %} 当前用户的首选语言，`示例：`en-us`。（请参阅*How Django discovers language preference*。）
- `{% get_current_language_bidi 为 LANGUAGE_BIDI %} 当前语言环境的方`如果为True，则为从右到左的语言，例如：希伯来语，阿拉伯语。如果为False，则为从左到右的语言，例如：英语，法语，德语等。

If you enable the `django.template.context_processors.i18n` context processor then each `RequestContext` will have access to `LANGUAGES`, `LANGUAGE_CODE`, and `LANGUAGE_BIDI` as defined above.

Changed in Django 1.8:

默认情况下，新项目的`i18n`上下文处理器未启用。

您还可以使用提供的模板标记和过滤器检索有关任何可用语言的信息。要获取有关单一语言的信息，请使用`标签： get_language_info %}`

```
{% get_language_info for LANGUAGE_CODE as lang %}
{% get_language_info for "pl" as lang %}
```

然后，您可以访问信息：

```
Language code: {{ lang.code }}<br />
Name of language: {{ lang.name_local }}<br />
Name in English: {{ lang.name }}<br />
Bi-directional: {{ lang.bidi }}
```

您还可以使用 `{% get_language_info_list %}` 模板标记来检索语言列表在 `LANGUAGES` 中指定的活动语言）。有关如何使用 `显示语言选择器的示例，请参阅[_the section about the set_language redir` 。

除了 `LANGUAGES` 样式嵌套元组之外， `{% get_language_info_list %}` 语言代码列表。如果你在你的观点：

```
context = {'available_languages': ['en', 'es', 'fr']}
return render(request, 'mytemplate.html', context)
```

您可以在模板中迭代这些语言：

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

还有简单的过滤器为了方便：

- `{{ LANGUAGE_CODE | language_name }}` （“德语”）
- `{{ LANGUAGE_CODE | language_name_local }}` （“Deutsch”）
- `{{ LANGUAGE_CODE | language_bidi }}` （假）

国际化：在JavaScript代码中

将翻译添加到JavaScript会带来一些问题：

- JavaScript代码无法访问 `gettext` 实现。
- JavaScript代码无权访问 `.po` 或 `.mo` 文件；它们需要由服务器交付。
- JavaScript的翻译目录应尽可能小。

Django为这些问题提供了一个集成的解决方案：它将翻译传递给JavaScript，因此您可以在JavaScript中调用 `gettext` 等。

的 `javascript_catalog`

`javascript_catalog` (*request*, *domain='djangojs'*, *packages=None*)[source]

这些问题的主要解决方案是 `django.views.i18n.javascript_catalog()` 视图，它发送一个JavaScript代码库，其函数模仿 `gettext` 接口，翻译字符串数组。这些翻译字符串取自应用程序或Django core，根据您在 `info_dict` 或URL中指定的内容。还包括 `LOCALE_PATHS` 中列出的路径。

你这样挂钩：

```
from django.views.i18n import javascript_catalog

js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = [
    url(r'^jsi18n/$', javascript_catalog, js_info_dict),
]
```

`packages` 中的每个字符串都应采用Python点分包格式（与 `INSTALLED_APPS` 中的字符串格式相同），并且应该指向包含 `locale` 如果指定多个包，则所有这些目录将合并到一个目录中。如果您的JavaScript使用来自不同应用程序的字符串，这将非常有用。

翻译的优先级是使得稍后在 `packages` 参数中出现的包比在开始处出现的包具有更高的优先级，这在针对相同文字冲突翻译的情况下是重要的。

默认情况下，视图使用 `djangojs` gettext域。这可以通过更改 `domain` 参数来更改。

您可以通过将包放入URL模式来使视图动态：

```
urlpatterns = [
    url(r'^jsi18n/(?P<packages>\S+?)/$', javascript_catalog),
]
```

这样，您可以将包指定为由URL中的“+”号分隔的包名称列表。如果您的网页使用来自不同应用程式的程式码，且这项变更经常发生，而且您不想提取一个大型目录档案，这项功能就特别实用。作为安全措施，这些值只能是 `django.conf` 或来自 `INSTALLED_APPS` 设置的任何包。

在 `LOCALE_PATHS` 设置中列出的路径中找到的JavaScript翻译也始终包括在内。为了与用于Python和模板的翻译查找顺序算法保持一致， `LOCALE_PATHS` 中列出的目录具有最高优先级，首先出现的优先级高于稍后出现的优先级。

使用JavaScript翻译目录

要使用目录，只需拉入动态生成的脚本，如下所示：

```
<script type="text/javascript" src="{% url 'django.views.i18n.ja
vascript_catalog' %}"></script>
```

这使用反向URL查找来查找JavaScript目录视图的URL。加载目录后，您的JavaScript代码可以使用以下方法：

- `gettext`
- `ngettext`
- `interpolate`
- `get_format`
- `gettext_noop`
- `pgettext`
- `npgettext`
- `pluralidx`

gettext

`gettext` 函数的行为与您的Python代码中的标准 `gettext` 接口类似：

```
document.write(gettext('this is to be translated'));
```

ngettext

`ngettext` 函数提供了一个多元化单词和短语的接口：

```
var object_count = 1 // or 0, or 2, or 3, ...
s = ngettext('literal for the singular case',
        'literal for the plural case', object_count);
```

插

`interpolate` 函数支持动态填充格式字符串。插值语法是从Python借用的，因此 `interpolate` 函数支持位置和命名插值：

- 位置插值： `obj` 包含一个JavaScript Array对象，其元素值然后按照它们出现的相同顺序在其相应的 `fmt` 占位符中顺序插值。例如：

  ```
  fmts = ngettext('There is %s object. Remaining: %s',
          'There are %s objects. Remaining: %s', 11);
  s = interpolate(fmts, [11, 20]);
  // s is 'There are 11 objects. Remaining: 20'
  ```

- Named interpolation: This mode is selected by passing the optional boolean `named` parameter as `true` . `obj` 包含JavaScript对象或关联数组。例如：

```
d = {
    count: 10,
    total: 50
};

fmts = ngettext('Total: %(total)s, there is %(count)s object
',
'there are %(count)s of a total of %(total)s objects', d.cou
nt);
s = interpolate(fmts, d, true);
```

你不应该使用字符串插值，但是这仍然是JavaScript，所以代码必须重复正则表达式替换。这不像Python中的字符串插值那么快，所以保持它在那些你真正需要它的情况下（例如，与 `ngettext` 一起产生正确的复数）。

get_format

`get_format` 函数可以访问配置的i18n格式设置，并可以检索给定设置名称的格式字符串：

```
document.write(get_format('DATE_FORMAT'));
// 'N j, Y'
```

它可以访问以下设置：

- 日期格式
- DATE_INPUT_FORMATS
- DATETIME_FORMAT
- DATETIME_INPUT_FORMATS
- DECIMAL_SEPARATOR
- FIRST_DAY_OF_WEEK
- MONTH_DAY_FORMAT
- NUMBER_GROUPING
- SHORT_DATE_FORMAT
- SHORT_DATETIME_FORMAT
- THOUSAND_SEPARATOR
- 时间格式
- TIME_INPUT_FORMATS
- YEAR_MONTH_FORMAT

这对于保持与Python呈现的值的格式一致性很有用。

gettext_noop

这模拟 `gettext` 函数，但什么都不做，返回任何传递给它的：

```
document.write(gettext_noop('this will not be translated'));
```

这对于剔除将来需要翻译的代码部分很有用。

pgettext

`pgettext` 函数的行为类似于Python变体（ `pgettext()` ），提供上下文翻译词：

```
document.write(pgettext('month name', 'May'));
```

npgettext

`npgettext` 函数的行为类似于Python变体（ `npgettext()` ），提供了一个多元化

```
document.write(npgettext('group', 'party', 1));
// party
document.write(npgettext('group', 'party', 2));
// parties
```

复数

`pluralidx` 函数以与 `pluralize` 模板过滤器类似的方式工作，确定给定的 `count` 是否应使用字的复数形式：

```
document.write(pluralidx(0));
// true
document.write(pluralidx(1));
// false
document.write(pluralidx(2));
// true
```

在最简单的情况下，如果不需要自定义复数，则对于所有其他数字，对于整数 `1` 和 `true` 返回 `false` 。

然而，在所有语言中，复数并不是这么简单。如果语言不支持复数，则提供空值。

此外，如果围绕复数有复杂的规则，目录视图将呈现条件表达式。这将评估 `true` （应为pluralize）或 `false` （应不 pluralize）值。

性能注意事项

`javascript_catalog()` 视图在每个请求中从 `.mo` 文件生成目录。由于它的输出是恒定的 - 至少对于给定版本的网站 - 它是缓存的一个很好的候选人。

服务器端缓存将减少CPU负载。它很容易用 `cache_page()` 装饰器实现。要在翻译更改时触发缓存无效，请提供版本相关的密钥前缀，如下面的示例所示，或者根据版本相关的网址映射视图。

```
from django.views.decorators.cache import cache_page
from django.views.i18n import javascript_catalog

# The value returned by get_version() must change when translati
ons change.
@cache_page(86400, key_prefix='js18n-%s' % get_version())
def cached_javascript_catalog(request, domain='djangojs', packag
es=None):
    return javascript_catalog(request, domain, packages)
```

客户端缓存可以节省带宽，并加快网站加载速度。如果您使用的是ETags（ `USE_ETAGS = True` ），则表示您已完成付款。否则，您可以应用*conditional decorators*。在以下示例中，无论何时重新启动应用程序服务器，缓存都将失效。

```
from django.utils import timezone
from django.views.decorators.http import last_modified
from django.views.i18n import javascript_catalog

last_modified_date = timezone.now()

@last_modified(lambda req, **kw: last_modified_date)
def cached_javascript_catalog(request, domain='djangojs', packag
es=None):
    return javascript_catalog(request, domain, packages)
```

您甚至可以预先生成JavaScript目录作为部署过程的一部分，并将其作为静态文件提供。这种激进的技术在django-statici18n中实现。

国际化：在URL模式中

Django提供了两种机制来国际化URL模式：

- 将语言前缀添加到网址格式的根目录，使 `LocaleMiddleware` 可以从请求的网址中检测要激活的语言。
- 使网址模式可通过 `django.utils.translation.ugettext_lazy()` 函数进行翻译。

警告

使用这些功能之一需要为每个请求设置活动语言；换句话说，您需要在 `MIDDLEWARE_CLASSES` 设置中设置 `django.middleware.locale.LocaleMiddleware` 。

URL模式中的语言前缀

`i18n_patterns` (*prefix*, *pattern_description*, *...*)[source]

自1.8版起已弃用： `i18n_patterns()` 的 `prefix` 参数已被弃用，不会在Django 2.0中受支持。只需传递 `django.conf.urls.url()` 实例的列表即可。

此函数可以在根URLconf中使用，Django会自动将当前活动语言代码添加到 `i18n_patterns()` 中定义的所有网址模式。网址格式示例：

```
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns

from about import views as about_views
from news import views as news_views
from sitemap.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap, name='sitemap_xml'),
]

news_patterns = [
    url(r'^$', news_views.index, name='index'),
    url(r'^category/(?P<slug>[\w-]+)/$', news_views.category, na
me='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail
'),
]

urlpatterns += i18n_patterns(
    url(r'^about/$', about_views.main, name='about'),
    url(r'^news/', include(news_patterns, namespace='news')),
)
```

定义这些网址格式后，Django会自动将语言前缀添加到由 `i18n_patterns` 函数添加的网址格式。例：

```
from django.core.urlresolvers import reverse
from django.utils.translation import activate

>>> activate('en')
>>> reverse('sitemap_xml')
'/sitemap.xml'
>>> reverse('news:index')
'/en/news/'

>>> activate('nl')
>>> reverse('news:detail', kwargs={'slug': 'news-slug'})
'/nl/news/news-slug/'
```

警告

`i18n_patterns()` 只能在根URLconf中使用。在包含的URLconf中使用它会引发 `ImproperlyConfigured` 异常。

警告

确保您没有可能与自动添加的语言前缀相冲突的非前缀网址格式。

翻译网址格式

网址格式也可以使用 `ugettext_lazy()` 函数标记为可翻译。例：

```
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns
from django.utils.translation import ugettext_lazy as _

from about import views as about_views
from news import views as news_views
from sitemaps.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap, name='sitemap_xml'),
]

news_patterns = [
    url(r'^$', news_views.index, name='index'),
    url(_(r'^category/(?P<slug>[\w-]+)/$'), news_views.category,
 name='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail
'),
]

urlpatterns += i18n_patterns(
    url(_(r'^about/$'), about_views.main, name='about'),
    url(_(r'^news/'), include(news_patterns, namespace='news')),
)
```

创建翻译后，`reverse()` 函数将返回活动语言的URL。例：

```
from django.core.urlresolvers import reverse
from django.utils.translation import activate

>>> activate('en')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/en/news/category/recent/'

>>> activate('nl')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/nl/nieuws/categorie/recent/'
```

警告

在大多数情况下，最好只在语言代码前缀的模式块中使用已翻译的URL（使用 `i18n_patterns()` ），以避免粗心大意的翻译URL导致与非翻译的网址格式。

在模板中反转

如果在模板中反转本地化URL，它们总是使用当前语言。要链接到其他语言的网址，请使用 `language` 模板标记。它在所包含的模板部分中启用给定的语言：

```
{% load i18n %}

{% get_available_languages as languages %}

{% trans "View this category in:" %}
{% for lang_code, lang_name in languages %}
    {% language lang_code %}
    <a href="{% url 'category' slug=category.slug %}">{{ lang_na
me }}</a>
    {% endlanguage %}
{% endfor %}
```

`language` 标签需要将语言代码作为唯一的参数。

本地化：如何创建语言文件

一旦应用程序的字符串文字已经标记为以后翻译，翻译本身需要被写入（或获得）。这是如何工作。

消息文件

第一步是为新语言创建*message file*。消息文件是纯文本文件，表示单一语言，包含所有可用的翻译字符串以及如何以给定语言表示它们。消息文件具有 `.po` 文件扩展名。

Django提供了一个工具， `django-admin makemessages` ，自动创建和维护这些文件。

Gettext实用程序

The `makemessages` command (and `compilemessages` discussed later) use commands from the GNU gettext toolset: `xgettext` , `msgfmt` , `msgmerge` and `msguniq` .

支持的 `gettext` 实用程序的最低版本为0.15。

要创建或更新消息文件，请运行以下命令：

```
django-admin makemessages -l de
```

...其中 `de` 是您要创建的消息文件的*locale name*。例如，巴西葡萄牙语为 `pt_BR` ，奥地利德语为 `de_AT` ，印尼语为 `id` 。

脚本应该从两个地方之一运行：

- Django项目的根目录（包含 `manage.py` 的目录）。
- 您的一个Django应用程序的根目录。

脚本运行在项目源代码树或应用程序源代码树中，并拉出所有标记为翻译的字符串（请参阅*How Django discovers translations*，并确保 `LOCALE_PATHS` 已正确配置）。它在目录 `locale/LANG/LC_MESSAGES` 中创建（或更新）消息文件。在 `de` 示例中，文件将为 `locale/de/LC_MESSAGES/django.po` 。

Changed in Django 1.7:

当从项目的根目录运行 `makemessages` 时，提取的字符串将自动分发到正确的消息文件。也就是说，从包含 `locale` 目录的应用的文件中提取的字符串将放在该目录下的消息文件中。从没有任何 `locale` 目录的应用的文件中提取的字符串将进入 `LOCALE_PATHS` 中列出的目录下的消息文件，否则会生成错误，如果 `LOCALE_PATHS` 为空。

默认情况下， `django-admin makemessages` 检查包含 `.html` 或 `.txt` 如果要覆盖该默认值，请使用 `--extension` 或 `-e` 选项指定要检查的文件扩展名：

```
django-admin makemessages -l de -e txt
```

使用逗号分隔多个扩展程序和/或使用 `-e` 或 `--extension` 多次：

```
django-admin makemessages -l de -e html,txt -e xml
```

警告

当*creating message files from JavaScript source code*创建消息文件时，您需要使用特殊的'djangojs'域，而不是 `-e js` 。

使用Jinja2模板？

`makemessages` 不能理解Jinja2模板的语法。要从包含Jinja2模板的项目中提取字符串，请改用Babel。

下面是一个示例 `babel.cfg` 的配置文件：

```
# Extraction from Python source files
[python: **.py]

# Extraction from Jinja2 templates
[jinja2: **.jinja]
extensions = jinja2.ext.with_
```

请确保您列出了您使用的所有扩展程序！否则Babel将无法识别这些扩展定义的标签，并会忽略包含它们的Jinja2模板。

Babel提供与 `makemessages` 类似的功能，可以替换它，而不依赖于 `gettext` 。有关详细信息，请阅读有关处理消息目录的文档。

没有gettext？

如果您没有安装 `gettext` 实用程序， `makemessages` 将创建空文件。如果是这种情况，请安装 `gettext` 实用程序，或只复制英文消息文件（ `locale/en/LC_MESSAGES/django.po` ）点；它只是一个空的翻译文件。

在Windows上工作？

如果您使用Windows并需要安装GNU gettext实用程序，因此 `makemessages` 可以工作，有关详细信息，请参阅*gettext on Windows*。

`.po` 文件的格式很简单。每个 `.po` 文件包含一小部分元数据，例如翻译维护者的联系信息，但文件的大部分是消息的列表 - 翻译字符串之间的简单映射和特定语言的实际翻译文本。

例如，如果您的Django应用程序包含文本 `“欢迎 到 我的 网站的翻译字符串” t4>` ，像这样：

```
_("Welcome to my site.")
```

... then `django-admin makemessages` 将创建一个包含以下代码片段的 `.po` 文件：

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

快速解释：

- `msgid` 是翻译字符串，它显示在源中。不要改变它。
- `msgstr` 是您放置语言特定翻译的位置。它开始是空的，所以你有责任改变它。确保您在翻译周围保留引号。
- 为了方便，每个消息以以 `#` 为前缀并位于 `msgid` 之上的注释行的形式包括翻译字符串所从的文件名和行号收集。

长消息是一种特殊情况。在那里，紧跟 `msgstr` （或 `msgid` ）之后的第一个字符串是空字符串。然后内容本身将被写在下几行作为每行一个字符串。这些字符串是直接连接的。不要忘记字符串中的尾随空格；否则，它们将被粘在一起，没有空格！

注意你的字符集

由于 `gettext` 工具的内部工作方式，因为我们要在Django的核心和应用程序中允许非ASCII源字符串，您必须使用UTF-8作为编码为您的PO文件（创建PO文件时的默认值）。这意味着每个人都将使用相同的编码，这在Django处理PO文件时很重要。

要对所有源代码和模板重新审核新的翻译字符串，并更新所有所有语言的邮件文件，请运行：

```
django-admin makemessages -a
```

编译消息文件

创建消息文件后 - 每次对其进行更改时，都需要将其编译为更有效的形式，以供 `gettext` 使用。使用 `django-admin compilemessages` 实用程序执行此操作。

此工具运行所有可用的 `.po` 文件，并创建 `.mo` 文件，这是优化为 `gettext` 使用的二进制文件。在运行 `django-admin makemessages` 的同一目录中，运行 `django-admin compilemessages` 像这样：

```
django-admin compilemessages
```

而已。您的翻译已准备就绪。

在Windows上工作？

If you're using Windows and need to install the GNU gettext utilities so `django-admin compilemessages` works see *gettext on Windows* for more information.

.po文件：编码和BOM使用。

Django只支持以UTF-8编码并且没有任何BOM（字节顺序标记）的 `.po` 文件，因此如果您的文本编辑器在默认情况下将这样的标记添加到文件的开头，则需要重新配置它。

从**JavaScript**源代码创建消息文件

您可以使用 `django-admin makemessages` 工具，以与其他Django邮件文件相同的方式创建和更新邮件文件。唯一的区别是，您需要通过提供 `-d 来明确指定在gettext中的内容是什么，在这种情况下为` djangojs `djangojs` 参数，如下所示：

```
django-admin makemessages -d djangojs -l de
```

这将创建或更新用于德语的JavaScript的消息文件。更新邮件文件后，只需以与对普通Django邮件文件相同的方式运行 `django-admin compilemessages` 即可。

gettext on Windows

这只适用于要提取消息ID或编译消息文件（ `.po` ）的人员。翻译工作本身只涉及编辑此类型的现有文件，但如果要创建自己的消息文件，或者想要测试或编译更改的消息文件，则需要使用 `gettext` 实用程序：

- 从GNOME服务器下载以下zip文件 https://download.gnome.org/binaries/win32/dependencies/
    - `gettext-runtime-X.zip`
    - `gettext-tools-X.zip`

  `X` 是版本号，我们需要 `0.15` 或更高版本。
- 将两个文件中的 `bin` 目录的内容提取到系统上相同的文件夹（即 C：Program Filesgettext-utils / t2>）
- 更新系统路径：
    - Control Panel > System > Advanced > Environment Var .
    - 在 `系统 变量` 列表中，单击 `Path` ，单击 `Edit` 。
    - 在 `变量 値结束时添加` ； C：Program Filesgettext-utilsin / t5>``字段。

只要 `xgettext - version` 命令正常工作，您还可以使用您在其他地方获得的 `gettext` 二进制。如果在命令 `xgettext - version` 中输入命令，则不要尝试使用 `gettext` Windows命令提示符导致弹出窗口说“xgettext.exe生成错误，将被Windows关闭”。

自定义 `makemessages`

如果要向 `xgettext` 传递附加参数，则需要创建自定义 `makemessages` 命令并覆盖其 `xgettext_options` 属性：

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):
    xgettext_options = makemessages.Command.xgettext_options + [
'--keyword=mytrans']
```

如果您需要更多灵活性，还可以向自定义 `makemessages` 命令中添加一个新参数：

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):

    def add_arguments(self, parser):
        super(Command, self).add_arguments(parser)
        parser.add_argument('--extra-keyword', dest='xgettext_ke
ywords',
                            action='append')

    def handle(self, *args, **options):
        xgettext_keywords = options.pop('xgettext_keywords')
        if xgettext_keywords:
            self.xgettext_options = (
                makemessages.Command.xgettext_options[:] +
                ['--keyword=%s' % kwd for kwd in xgettext_keywor
ds]
            )
        super(Command, self).handle(*args, **options)
```

杂

的 `set_language`

`set_language` (*request*)[source]

方便起见, Django自带了一个, `django.views.i18n.set_language()` 视图, 作用是设置用户语言偏好并重定向返回到前一页面

在URLconf中加入下面这行代码来激活这个视图：

```
url(r'^i18n/', include('django.conf.urls.i18n')),
```

(注意这个例子使得这个视图在 `/i18n/setlang/` 中有效.)

警告

请确保您不要在 `i18n_patterns()` 中包含上述网址 - 它需要与语言无关才能正常工作。

该视图需要通过 `POST` 方法调用，并在请求中设置 `language` 参数。如果启用了会话支持，则视图会在用户的会话中保存语言选择。否则，它会将语言选项保存在默认名为 `django_language` 的Cookie中。（名称可以通过 `LANGUAGE_COOKIE_NAME` 设置更改。）

设置语言选择后，Django重定向用户，遵循此算法：

- Django在 `POST` 数据中查找 `next` 参数。
- 如果不存在或为空，Django会尝试 `Referrer` 标头中的网址。
- 如果这是空的 - 例如，如果用户的浏览器禁止该标题 - 那么用户将被重定向到 `/` （网站根）作为后备。

这里是HTML模板代码的示例：

```
{% load i18n %}
<form action="{% url 'set_language' %}" method="post">
{% csrf_token %}
<input name="next" type="hidden" value="{{ redirect_to }}" />
<select name="language">
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_language_info_list for LANGUAGES as languages %}
{% for language in languages %}
<option value="{{ language.code }}"{% if language.code == LANGUA
GE_CODE %} selected="selected"{% endif %}>
    {{ language.name_local }} ({{ language.code }})
</option>
{% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

在此示例中，Django在 `redirect_to` 上下文变量中查找用户将被重定向到的网页的网址。

明确设定使用语言

您可能需要显式设置当前会话的活动语言。例如，可能从另一个系统检索用户的语言偏好。您已介绍过 `django.utils.translation.activate()` 。这只适用于当前线程。要为整个会话持久保存语言，还需修改会话中的 `LANGUAGE_SESSION_KEY` ：

```
from django.utils import translation
user_language = 'fr'
translation.activate(user_language)
request.session[translation.LANGUAGE_SESSION_KEY] = user_languag
e
```

您通常希望同时使用： `django.utils.translation.activate()` 将更改此线程的语言，并修改会话使此首选项在未来的请求中保持不变。

如果您没有使用会话，该语言将保留在Cookie中，其名称在 `LANGUAGE_COOKIE_NAME` 中配置。例如：

```
from django.utils import translation
from django import http
from django.conf import settings
user_language = 'fr'
translation.activate(user_language)
response = http.HttpResponse(...)
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, user_language
)
```

使用视图和模板外部的翻译

虽然Django提供了一组丰富的i18n工具以用于视图和模板，但它并不限制对Django特定代码的使用。Django翻译机制可以用于将任意文本翻译成Django支持的任何语言（当然，只要存在适当的翻译目录）。您可以加载翻译目录，将其激活并将文本翻译为您选择的语言，但请记住切换回原始语言，因为激活翻译目录是基于每个线程完成的，这样的更改将影响在同一个线程中运行的代码。

例如：

```
from django.utils import translation

def welcome_translated(language):
    cur_language = translation.get_language()
    try:
        translation.activate(language)
        text = translation.ugettext('welcome')
    finally:
        translation.activate(cur_language)
    return text
```

无论 `LANGUAGE_CODE` 和中间件设置的语言如何，使用值“de”调用此函数将给予 `"Willkommen"`

特别感兴趣的函数是 `django.utils.translation.get_language()` ，它返回当前线程中使用的语言 `django.utils.translation.activate()` 用于当前线程的翻译目录，以及用于检查Django是否支持给定语言的 `django.utils.translation.check_for_language()` 。

为了帮助编写更简洁的代码，还有一个上下文管理器 `django.utils.translation.override()` ，用于在输入时存储当前语言，并在退出时恢复。有了它，上面的例子变成：

```
from django.utils import translation

def welcome_translated(language):
    with translation.override(language):
        return translation.ugettext('welcome')
```

语言cookie

可以使用多种设置来调整语言Cookie选项：

- LANGUAGE_COOKIE_NAME

New in Django 1.7.

- LANGUAGE_COOKIE_AGE
- LANGUAGE_COOKIE_DOMAIN
- LANGUAGE_COOKIE_PATH

实现细节

Django 翻译的特点

Django 的翻译机制使用Python 自带的 `gettext` 模块。如果你了解 `gettext` ，你应该注意到Django 翻译方式的这些特点：

- 字符串域是 `django` 或 `djangojs` 。这个字符串域用于区分不同的而程序，这些程序将它们的数据保存在共同的消息文件库中（通常是 `/usr/share/locale/` ）。 `django` 域用于Python 和模板的翻译字符串，而且还会加载到全局翻译目录中。 `djangojs` 域只用于JavaScript 的翻译目录，以确保它尽可能的小。
- Django 没有单独使用 `xgettext` 。它使用Python 对 `xgettext` 和 `msgfmt` 进行的封装。最主要是为了方便。

Django 如何发现语言偏好

一旦你准备好翻译，或者你只是想使用Django 自带的翻译，你需要为你的应用启用翻译。

在后台，Django 有一个非常灵活的模型决定应该使用哪种语言 —— 所有用户还是特定的用户，或者两种都可以。

要设置安装范围的语言首选项，请设置 `LANGUAGE_CODE` 。Django使用这种语言作为默认翻译 - 如果没有找到更好的匹配翻译通过场所中间件（见下文）使用的方法之一的最终尝试。

如果你想要的是用你的母语运行Django，你只需设置 `LANGUAGE_CODE` 并确保相应的*message files*及其编译版本（ `.mo` ）。

如果你想让每个用户指定首选语言，那么你还要 使用 `LocaleMiddleware` . `LocaleMiddleware` 会打开基于请求数据的语言选择。它为每个人用户的内容进行个性化。

要使用 `LocaleMiddleware` ，请将 `'django.middleware.locale.LocaleMiddleware'` 添加到您的 `MIDDLEWARE_CLASSES` 设置。由于中间件顺序很重要，因此您应该遵循以下准则：

- 确保它是安装的第一个中间件之一。
- 它应该在 `SessionMiddleware` 之后，因为 `LocaleMiddleware` 使用会话数据。它应该在 `CommonMiddleware` 之前，因为 `CommonMiddleware` 需要一个激活的语言才能解析请求的URL。
- 如果使用 `CacheMiddleware` ，请在其后放置 `LocaleMiddleware` 。

例如，你的 `MIDDLEWARE_CLASSES` 也许看起来像这样子

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
)
```

（有关中间件的详情，请参阅*middleware documentation*。）

`LocaleMiddleware` 尝试通过以下算法确定用户的语言首选项：

- 首先，它在请求的URL中查找语言前缀。这仅在您在根URLconf中使用 `i18n_patterns` 函数时才会执行。有关语言前缀以及如何将网址格式国际化的详细信息，请参见*Internationalization: in URL patterns*中。

- 否则，它会在当前用户的会话中查找 `LANGUAGE_SESSION_KEY` 键。

  Changed in Django 1.7:

  在以前的版本中，键名为 `django_language` ，并且 `LANGUAGE_SESSION_KEY` 常数不存在。

- 如果没有，它寻找一个cookie。

  所使用的Cookie名称由 `LANGUAGE_COOKIE_NAME` 设置设置。（默认名称为 `django_language` 。）

- 如果没有，它会查看 `Accept-Language` HTTP标头。此标题由您的浏览器发送，并按优先级顺序告诉服务器您喜欢哪种语言。Django尝试标题中的每种语言，直到找到一个可用的翻译。

- 否则，它使用全局 `LANGUAGE_CODE` 设置。

笔记：

- 在每个位置，语言首选项应为标准*language format*，作为字符串。例如，巴西葡萄牙语是 `pt-br` 。
- 如果基本语言可用，但指定的子语言不是，Django使用基本语言。例如，如果用户指定 `de-at` （奥地利德语），但Django只有 `de` 可用，则Django使用 `de` 。
- 只能选择 `LANGUAGES` 设置中列出的语言。如果要将语言选择限制为所提供语言的一个子集（因为您的应用程序不提供所有这些语言），请将 `LANGUAGES` 设置为语言列表。例如：

  ```
  LANGUAGES = (
    ('de', _('German')),
    ('en', _('English')),
  )
  ```

  此示例将可用于自动选择的语言限制为德语和英语（以及任何子语言，如de-ch或en-us）。

- 如果您定义自定义 `LANGUAGES` 设置（如上一个项目符号所述），则可以将语言名称标记为翻译字符串 - 但使用 `ugettext_lazy()` 而不是 `ugettext()` 以避免循环导入。

  下面是一个示例设置文件：

  ```
  from django.utils.translation import ugettext_lazy as _

  LANGUAGES = (
      ('de', _('German')),
      ('en', _('English')),
  )
  ```

一旦 `LocaleMiddleware` 确定用户的首选项，就会将此首选项设置为 `request.` 每个 `HttpRequest` 的LANGUAGE_CODE。随意在您的视图代码中读取此值。这里有一个简单的例子：

```
from django.http import HttpResponse

def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.
")
    else:
        return HttpResponse("You prefer to read another language
.")
```

请注意，使用静态（无中间件）翻译时，语言位于 `settings.` LANGUAGE_CODE，而在动态（中间件）翻译时，它在 `request.` LANGUAGE_CODE。

Django是如何找到翻译文件的

在运行时，Django构建一个内存中的文字 - 翻译目录。为了实现这一点，它通过遵循该算法关于其检查不同文件路径以加载编译的*message files*（ `.mo` ）和优先级的顺序来寻找翻译多个翻译为同一个字面量：

1. `LOCALE_PATHS` 中列出的目录具有最高优先级，首先出现的优先级高于稍后出现的优先级。
2. 然后，它会查找并使用 `INSTALLED_APPS` 中列出的每个已安装应用程序中是否存在 `locale` 目录。首先出现的优先级高于稍后出现的优先级。
3. 最后，在 `django/conf/locale` 中提供的Django提供的基本翻译用作后备。

也可以看看

JavaScript资源中包含的文字的翻译是根据类似但不完全相同的算法查找的。有关详细信息，请参阅*javascript_catalog view documentation*。

在所有情况下，包含翻译的目录的名称应使用*locale name*符号命名。例如。 `de` ， `pt_BR` ， `es_AR` 等。

这样，您可以编写包含自己的翻译的应用程序，并且可以覆盖项目中的基本翻译。或者，您可以从几个应用程序构建一个大项目，并将所有翻译成一个大的通用消息文件，特定于您正在撰写的项目。这是你的选择。

所有消息文件存储库的结构都是相同的。他们是：

- 搜索设置文件中 `LOCALE_PATHS` 中列出的所有路径 `<language>/LC_MESSAGES/django.(` po | mo）
- `$APPPATH/locale/<language>/LC_MESSAGES/django.(` po | mo）
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/djan` po | mo）

要创建邮件文件，请使用 `django-admin makemessages` 工具。And you use `django-admin compilemessages` to produce the binary `.mo` files that are used by `gettext`.

您还可以运行 `django-admin compilemessages --settings=path.to.settings` 您的 `LOCALE_PATHS` 设置中的所有目录。

格式本地化

概览

Django的格式化系统可以在模板中使用当前地区特定的格式，来展示日期、时间和数字。也可以处理表单中输入的本地化。

当它被开启时，访问相同内容的两个用户可能会看到以不同方式格式化的日期、时间和数字，这取决于它们的当前地区的格式。

格式化系统默认是禁用的。需要在你的设置文件中设置 `USE_L10N = True` 来启用它。

注意

为了方便起见，`django-admin startproject创建的默认的` settings.py `文件包含了 ` [USE_L10N = True](../../ref/settings.html#std:setting-USE_L10N) 的设置。 但是要注意，要开启千位分隔符的数字格式化，你需要在你的设置文件中设置 `USE_THOUSAND_SEPARATOR = True` 。或者，你也可以在你的模板中使用 `intcomma` 来格式化数字。

注意

`USE_I18N` 是另一个独立的并且相关的设置，它控制着Django是否应该开启翻译。详见翻译。

表单中的本地化识别输入

格式化开启之后，Django可以在表单中使用本地化格式来解析日期、时间和数字。也就是说，在表单上输入时，它会尝试不同的格式和地区来猜测用户使用的格式。

注意

Django对于展示数据，使用和解析数据不同的格式。尤其是，解析日期的格式不能使用 `%a` （星期名称的缩写）， `%A` （星期名称的全称）， `%b` （月份名称的缩写）， `%B` （月份名称的全称），或者 `%p` （上午/下午）。

只是使用 `localize` 参数，就能开启表单字段的本地化输入和输出：

```
class CashRegisterForm(forms.Form):
    product = forms.CharField()
    revenue = forms.DecimalField(max_digits=4, decimal_places=2,
localize=True)
```

在模板中控制本地化

当你使用 `USE_L10N` 来开启格式化的时候，Django会尝试使用地区特定的格式，无论值在模板的什么位置输出。

然而，这对于本地化的值不可能总是十分合适，如果你在输出JavaScript或者机器阅读的XML，你会想要使用去本地化的值。你也可能想只在特定的模板中使用本地化，而不是任何位置都使用。

DJango提供了 `l10n` 模板库，包含以下标签和过滤器，来实现对本地化的精细控制。

模板标签

localize

在包含的代码块内开启或关闭模板变量的本地化。

这个标签可以对本地化进行比 `USE_L10N` 更加精细的操作。

这样做来为一个模板激活或禁用本地化：

```
{% load l10n %}

{% localize on %}
    {{ value }}
{% endlocalize %}

{% localize off %}
    {{ value }}
{% endlocalize %}
```

注意

在 `{% localize %}` 代码块内并不遵循f `USE_L10N` 的值。

对于在每个变量基础上执行相同工作的模板过滤器，参见 `localize` 和 `unlocalize` 。

模板过滤器

localize

强制单一值的本地化。

例如：

```
{% load l10n %}

{{ value|localize }}
```

使用 `unlocalize` 来在单一值上禁用本地化。使用 `localize` 模板标签来在大块的模板区域内控制本地化。

unlocalize

强制单一值不带本地化输出。

例如：

```
{% load l10n %}

{{ value|unlocalize }}
```

使用 `localize` 来强制单一值的本地化。使用 `localize` 模板标签来在大块的模板区域内控制本地化。

创建自定义的格式文件

Django为许多地区提供了格式定义，但是有时你可能想要创建你自己的格式，因为你的的确并没有现成的格式文件，或者你想要覆写其中的一些值。

Changed in Django 1.8:

添加了指定 `FORMAT_MODULE_PATH` 为列表的功能。之前只支持单一的字符串值。

指定你首先放置格式文件的位置来使用自定义格式。把你的 `FORMAT_MODULE_PATH` 设置设置为格式文件存在的包名来使用它，例如：

```
FORMAT_MODULE_PATH = [
    'mysite.formats',
    'some_app.formats',
]
```

文件并不直接放在这个目录中，而是放在和地区名称相同的目录中，文件也必须名为 `formats.py` 。

需要这样一个结构来自定义英文格式：

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

其中 `formats.py` 包含自定义的格式定义。例如：

```
from __future__ import unicode_literals

THOUSAND_SEPARATOR = '\xa0'
```

使用非间断空格(Unicode `00A0`)作为千位分隔符，来代替英语中默认的逗号。

提供本地化格式的限制

一些地区对数字使用上下文敏感的格式，Django的本地化系统不能自动处理它。

瑞士(德语)

瑞士的数字格式化取决于被格式化的数字类型。对于货币值，使用逗号作为千位分隔符，以及使用小数点作为十进制分隔符。对于其它数字，逗号用于十进制分隔符，空格用于千位分隔符。Django提供的本地格式使用通用的分隔符，即逗号用于十进制分隔符，空格用于千位分隔符。

> 译者：Django 文档协作翻译小组，原文：ocalized Web UI formatting and form input。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

"本地特色"附加功能

由于历史因素，Django自带了 `django.contrib.localflavor` -- 各种各样的代码片段，有助于在特定的国家地区或文化中使用。为了便于维护以及减少Django代码库的体积，这些代码现在在Django之外单独发布。

详见官方文档：

> https://django-localflavor.readthedocs.org/

这些代码托管在GIthub上面，https://github.com/django/django-localflavor。

如何迁移

如果你使用了老版本的 `django.contrib.localflavor` 包，或者 `django-localflavor-*` 的模板之一，执行这两个简单的步骤就可以更新你的代码：

- 在PyPI中安装第三方的 `django-localflavor` 包。
- 修改你应用的导入语句来引用新的包。

  例如，将：

  ```
  from django.contrib.localflavor.fr.forms import FRPhoneNumberField
  ```

  ...改为：

  ```
  from localflavor.fr.forms import FRPhoneNumberField
  ```

新的包中的代码和以前一样(它是直接从Django中复制出来的)，所以你并不用担心功能上的向后兼容问题。只需要修改导入语句。

弃用政策

在 Django 1.5中，导入 `django.contrib.localflavor` 会产生 `DeprecationWarning` 异常。也就是说你的代码还可以继续工作，但是你应该尽快修改它。

在Django 1.6中，导入 `django.contrib.localflavor` 将不会继续工作。

> 译者：Django 文档协作翻译小组，原文：“Local flavor”。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

时区

概览

当时区支持开启时，Django将时间用UTC格式存储到数据库中，在内部使用时区相关的对象，并且在模板(templates)与表单(forms)中将时间转换为终端用户所在时区的时间

当你的用户生活在多个时区，并且你希望根据他们所在的位置显示当地时间时很有用

即便你的网站仅能在一个时区内访问，在数据库中存储UTC 时间依然一种很好的做法。一个主要的原因是夏令时。 许多国家都拥有自己的一套夏令时系统，在这套系统里，春季的时间会提前，而秋季的时间便会后延。如果你只以当前时间为标准来开发，每年都会因为夏令时而引起两次错误（pytz文档更详细地讨论了这些问题。）这个对于你的博客可能没有什么影响，但是如果涉及到按年，按月，按小时来收费的话，那么就会是一个问题，解决这个问题的方法便是在代码中使用UTC时间，仅在与最终用户进行交互的时候使用本地时间。

Django默认关闭时区支持，如欲开启时区支持，则需在settings中设置 `USE_TZ = True` 。强烈推荐安装 pytz , 但根据所使用的数据库引擎、操作系统及时区的不同并不强制安装。如果在查询日期或时间时出现异常，请在检查bug之前先尝试安装pytz。可执行下述命令安装pytz：

```
$ pip install pytz
```

注意：

为方便起见，在由 `django-admin startproject` 创建的 `settings.py` 文件中已设置 `USE_TZ = True` 。

注意：

另外在settings中，还有一个 `USE_L10N` 设置选项，使用它可控制Django是否激活格式本地化。更多细节请参见格式本地化。

如果你正为某个与时区相关的特殊问题而纠结, 请阅读时区*FAQ*。

概念

Naive和aware类型的datetime对象

Python的 `datetime.datetime` 对象有一个 `tzinfo` 属性，该属性是 `datetime.tzinfo` 子类的一个实例，它被用来存储时区信息。 当某个datetime对象的tzinfo属性被设置并给出一个时间偏移量时，我们称该datetime对象是 **aware**（已知）的。否则称其为 **naive**（原生）的。

可用 `is_aware()` 和 `is_naive()` 函数来判断某个datetime对象是aware类型或naive类型。

当关闭时区支持时（USE_TZ=False），Django使用原生的datetime对象保存本地时间。在许多应用中，这是最简单的方式，并且足以满足要求。在这种情况下，可使用下列代码获取当前时间：|

```
import datetime

now = datetime.datetime.now()
```

当开启时区支持时 (`USE_TZ=True`), Django使用已知（aware）的datetime对象存储本地时间。如果在代码中创建了datetime对象, 那么它们也应该是aware类型的datetime对象。在此情况下，上述例子变成：

```
from django.utils import timezone

now = timezone.now()
```

警告：

对aware类型的datetime对象的处理并非总是非常直观。例如：对DST时区来说，标准datetime对象构造器的 `tzinfo` 参数并不能可靠地工作。此时，使用UTC一般来说是安全的；如果你的项目使用了其他的时区，那么应该认真阅读pytz的文档。

注意：

Python的 `datetime.time` 对象也具有包含一个 `tzinfo` 属性的特点，并且在PostgreSQL数据库引擎中，有一个 `time with time zone` 类型与该属性相匹配。但是，正如PostgreSQL的文档所指出的那样，该类型 “禁止使用那些可能导致出现问题的属性“。

Django仅支持naive(原生)类型的time对象， 并且，当试图保存一个aware类型的time对象时将会触发一个异常。这是因为，对于时区来说，如果保存的时间没有带有相关日期信息那就没有什么意义。.

naive datetime 对象的解释

当 `USE_TZ` 为 `True` , Django 为了保持向后兼容性依旧接受 naive datetime 对象. 当数据库层收到一个数据库时，它会尝试通过在*default time zone*中解释它来提醒它，并发出警告。

不幸的是，在DST转换期间，一些数据时间不存在或不明确。在这种情况下，pytz引发异常。其他 `tzinfo` 实施（例如未安装pytz时用作回退的本地时区）可能会引发异常或返回不准确的结果。这就是为什么当启用时区支持时，您应该始终创建感知datetime对象。

在实践中，这很少是一个问题。Django在模型和表单中给出了清晰的datetime对象，通常，通过 `timedelta` 算法从现有对象创建新的datetime对象。在应用程序代码中经常创建的唯一日期时间是当前时间，`timezone.now()` 自动执行正确的操作。

默认时区和当前时区

默认时区是由 `TIME_ZONE` 设置定义的时区。

当前时区是用于呈现的时区。

您应该使用 `activate()` 将当前时区设置为最终用户的实际时区。否则，将使用默认时区。

注意

如在 `TIME_ZONE` 的文档中所解释的，Django设置环境变量，以使其进程在默认时区运行。无论 `USE_TZ` 的值和当前时区的值如何，都会发生这种情况。

当 `USE_TZ` 为 `True` 时，这对于仍然依赖于本地时间的应用程序保持向后兼容性很有用。但是，*as explained above*，这不是完全可靠的，并且您应该始终在自己的代码中使用UTC中的感知数据时间。例如，使用 `utcfromtimestamp()` 而不是 `fromtimestamp()` - 不要忘记将 `tzinfo` 设置为 `utc`

选择当前时区

当前时区等效于当前*locale*的翻译。但是，没有等效的Django可以用来自动确定用户的时区的 `Accept-Language` HTTP标头。相反，Django提供*time zone selection functions*。使用它们来构建对您有意义的时区选择逻辑。

大多数关心时区的网站只是询问用户居住的时区，并将此信息存储在用户的配置文件中。对于匿名用户，他们使用其主要受众群体或UTC的时区。pytz提供助手，例如每个国家/地区的时区列表，您可以使用它预先选择最可能的选择。

下面是一个将当前时区存储在会话中的示例。（为了简单起见，它完全跳过错误处理。）

将以下中间件添加到 `MIDDLEWARE_CLASSES` ：

```
import pytz

from django.utils import timezone

class TimezoneMiddleware(object):
    def process_request(self, request):
        tzname = request.session.get('django_timezone')
        if tzname:
            timezone.activate(pytz.timezone(tzname))
        else:
            timezone.deactivate()
```

创建可以设置当前时区的视图：

```
from django.shortcuts import redirect, render

def set_timezone(request):
    if request.method == 'POST':
        request.session['django_timezone'] = request.POST['timez
one']
        return redirect('/')
    else:
        return render(request, 'template.html', {'timezones': py
tz.common_timezones})
```

在 `template.html` 中包含一个表单，它会 `POST` 到此视图：

```
{% load tz %}
{% get_current_timezone as TIME_ZONE %}
<form action="{% url 'set_timezone' %}" method="POST">
    {% csrf_token %}
    <label for="timezone">Time zone:</label>
    <select name="timezone">
        {% for tz in timezones %}
        <option value="{{ tz }}"{% if tz == TIME_ZONE %} selecte
d="selected"{% endif %}>{{ tz }}</option>
        {% endfor %}
    </select>
    <input type="submit" value="Set" />
</form>
```

表单中的时区感知输入

启用时区支持时，Django解释在*current time zone*中以表单形式输入的数据时间，并返回 `cleaned_data` 中的已知datetime对象。

如果当前时区对于不存在或由于它们落在DST转换（由pytz提供的时区）执行此操作而不存在或不明确的数据时间引发异常，则此类数据时间将报告为无效值。

模板中的时区感知输出

启用时区支持时，Django在模板中呈现时，将知道的datetime对象转换为*current time zone*。这与*format localization*。

警告

Django不转换天真的datetime对象，因为它们可能是模糊的，并且因为您的代码不应该产生幼稚的数据时间，当时区支持启用。但是，您可以使用下面描述的模板过滤器强制转换。

转换为本地时间并不总是适当的 - 您可能正在为计算机而不是为人类生成输出。以下由 `tz` 模板标记库提供的过滤器和标记允许您控制时区转换。

模板标签

当地时间

启用或禁用将已知datetime对象转换为包含块中的当前时区。

对于模板引擎，此标记与 `USE_TZ` 设置具有完全相同的效果。它允许更细粒度的转换控制。

要激活或取消激活模板块的转换，请使用：

```
{% load tz %}

{% localtime on %}
    {{ value }}
{% endlocaltime %}

{% localtime off %}
    {{ value }}
{% endlocaltime %}
```

注意

在 `{% 本地时间 %} t3内不遵守[` USE_TZ `](../../ref/settings.html#std:se`

时区

设置或取消所包含块中的当前时区。当前时区未设置时，将应用默认时区。

```
{% load tz %}

{% timezone "Europe/Paris" %}
    Paris time: {{ value }}
{% endtimezone %}

{% timezone None %}
    Server time: {{ value }}
{% endtimezone %}
```

get_current_timezone

您可以使用 `get_current_timezone` 标记获取当前时区的名称：

```
{% get_current_timezone as TIME_ZONE %}
```

如果您启用 `django.template.context_processors.tz` 上下文处理器，则每个 `RequestContext` 将包含一个 `TIME_ZONE` 变量，值为 `get_current_timezone()` 。

模板过滤器

这些过滤器接受意识和天真的数据时间。出于转换目的，他们假设幼稚的数据时间在默认时区。它们总是返回感知的数据时间。

当地时间

强制将单个值转换为当前时区。

例如：

```
{% load tz %}

{{ value|localtime }}
```

世界标准时间

强制将单个值转换为UTC。

例如：

```
{% load tz %}

{{ value|utc }}
```

时区

强制将单个值转换为任意时区。

参数必须是 `tzinfo` 子类的实例或时区名称。如果是时区名称，则需要pytz。

例如：

```
{% load tz %}

{{ value|timezone:"Europe/Paris" }}
```

迁移指南

以下是如何迁移在Django支持的时区之前启动的项目。

数据库

PostgreSQL

PostgreSQL后端将数据时间存储为 `timestamp 与 时间 区域` 。实际上，这意味着它将数据时间从连接的时区转换为存储上的UTC，以及从UTC转换为检索时的连接的时区。

因此，如果您使用PostgreSQL，您可以在 `USE_TZ = False` 和 `USE_TZ = True` 。数据库连接的时区将分别设置为 `TIME_ZONE` 或 `UTC` ，以便Django在所有情况下都获得正确的数据时间。您不需要执行任何数据转换。

其他数据库

其他后端存储没有时区信息的数据时间。如果您从 `USE_TZ = False` 切换到 `USE_TZ = True` ，您必须将您的数据从本地时间转换为UTC - 如果您的当地时间有DST，这是不确定的。

码

第一步是在您的设置文件中添加 `USE_TZ = True` ，然后安装pytz t5 >（如果可能）。在这一点上，事情应该主要工作。如果你在你的代码中创建天真的datetime对象，Django使他们知道在必要时。

但是，这些转换可能会在DST转换周围失败，这意味着您没有得到时区支持的全部好处。此外，你可能会遇到一些问题，因为它是不可能比较一个天真的datetime与意识datetime。由于Django现在给你知道的数据时间，你会得到异常，无论你比较来自一个模型或表单的日期时间与您在代码中创建的天真datetime。

所以第二步是重构你的代码，无论你实例化datetime对象，让他们知道。这可以递增地完成。 `django.utils.timezone` defines some handy helpers for compatibility code: `now()` , `is_aware()` , `is_naive()` , `make_aware()` , and `make_naive()` .

最后，为了帮助您找到需要升级的代码，当您尝试将天真的datetime保存到数据库时，Django会发出警告：

```
RuntimeWarning: DateTimeField ModelName.field_name received a na
ive
datetime (2012-01-01 00:00:00) while time zone support is active
.
```

在开发期间，您可以将此类警告转换为异常，并通过将以下内容添加到设置文件中获取回溯：

```
import warnings
warnings.filterwarnings(
        'error', r"DateTimeField .* received a naive datetime",
        RuntimeWarning, r'django\.db\.models\.fields')
```

夹具

当序列化意识datetime时，包括UTC偏移，像这样：

```
"2011-09-01T13:20:30+03:00"
```

对于天真的datetime，它显然不是：

```
"2011-09-01T13:20:30"
```

对于具有 `DateTimeField` 的模型，此差异使得无法编写支持和不支持时区支持的灯具。

使用 `USE_TZ = False` 或在Django 1.4之前生成的灯具使用“naive”格式。如果您的项目包含此类灯具，则在启用时区支持后，在加载时会看到 `RuntimeWarning` 。要摆脱警告，你必须将你的灯具转换为“意识”的格式。

您可以使用 `loaddata` ，然后 `dumpdata` 重新生成灯具。或者，如果它们足够小，您可以简单地编辑它们，以将与您的 `TIME_ZONE` 匹配的UTC偏移量添加到每个序列化的日期时间。

常问问题

建立

1. 我不需要多个时区。 我应该启用时区支持吗？

   是。当启用时区支持时，Django使用更准确的本地时间模型。这将屏蔽您在夏令时（DST）转换周围的微妙和不可再现的错误。

   在这方面，时区与Python中的 `unicode` 相当。起初很难。您得到编码和解码错误。然后你学习规则。并且一些问题消失 - 当您的应用程序接收到非ASCII输入时，您从不会再次遇到错误输出。

   当您启用时区支持时，您会遇到一些错误，因为您使用天真的数据时间，其中Django期望感知的数据时间。这样的错误显示在运行测试时，它们很容易修复。您将快速了解如何避免无效操作。

   另一方面，由于缺乏时区支持而导致的错误很难预防，诊断和修复。任何涉及计划任务或日期时间算法的事情都是一个微妙的bug，每年只会咬一次或一两次。

   因为这些原因，默认情况下在新项目中启用时区支持，并且您应该保留它，除非您有非常好的理由不要。

2. 我已启用时区支援。 我安全吗？

   也许。你更好地保护免受DST相关的错误，但你仍然可以通过不经意地将幼稚的数据时间转换为感知的数据时间来拍摄自己，反之亦然。

   如果您的应用程序连接到其他系统 - 例如，如果它查询Web服务 - 确保正确指定了数据时间。要安全地传输数据时间，它们的表示应该包括UTC偏移，或者它们的值应该是UTC（或两者都是！）。

   最后，我们的日历系统包含计算机的有趣陷阱：

```
&gt;&gt;&gt; import datetime
&gt;&gt;&gt; def one_year_before(value):       # DON'T DO THAT!
...     return value.replace(year=value.year - 1)
&gt;&gt;&gt; one_year_before(datetime.datetime(2012, 3, 1, 10, 0))
datetime.datetime(2011, 3, 1, 10, 0)
&gt;&gt;&gt; one_year_before(datetime.datetime(2012, 2, 29, 10, 0))
Traceback (most recent call last):
...
ValueError: day is out of range for month
```

（要实现此功能，您必须决定2012-02-29减去一年是2011-02-28还是2011-03-01，具体取决于您的业务需求。）

3. 我应该安装**pytz**吗？

   是。Django有一个不需要外部依赖的策略，因为pytz是可选的。但是，安装它更安全。

   一旦激活时区支持，Django需要定义默认时区。当pytz可用时，Django从tz数据库加载此定义。这是最准确的解决方案。否则，它依赖于操作系统报告的本地时间和UTC之间的差异来计算转换。这不太可靠，特别是在DST转换周围。

   此外，如果你想支持用户在多个时区，pytz是时区定义的参考。

故障排除

1. 我的应用程序式与发生当机 `TypeError： 无法 比较 offset-naive 和 偏移感知 数据时间` - 出了什么问题？

   让我们通过比较一个naive和一个意识datetime重现这个错误：

```
&gt;&gt;&gt; import datetime
&gt;&gt;&gt; from django.utils import timezone
&gt;&gt;&gt; naive = datetime.datetime.utcnow()
&gt;&gt;&gt; aware = timezone.now()
&gt;&gt;&gt; naive == aware
Traceback (most recent call last):
...
TypeError: can't compare offset-naive and offset-aware datetimes
```

   如果你遇到这个错误，很可能你的代码是比较这两个东西：

   - 由Django提供的datetime - 例如，从表单或模型字段读取的值。由于您启用时区支持，它意识到。
   - 你的代码生成的日期时间，这是天真的（或者你不会读这个）。

通常，正确的解决方案是更改您的代码，以使用意识到的datetime。

如果您编写的可插拔应用程序预期独立于 `USE_TZ` 的值工作，您可能会发现 `django.utils.timezone.now()` 很实用。当 `USE_TZ = False` 时，此函数将当前日期和时间返回为原始日期时间， `USE_TZ = True` 。您可以根据需要添加或减少 `datetime.timedelta` 。

2. 我看到很多 `RuntimeWarning： DateTimeField 收到 a naive datetime ``（YYYY-M` - 是不是？

   当启用时区支持时，数据库层希望从代码中仅接收知道的数据时间。当它收到一个天真的datetime时，会发生此警告。这表示您尚未完成移植您的代码以支持时区。有关此过程的提示，请参阅*migration guide*。

   在此期间，为了向后兼容，datetime被认为是在默认时区，这通常是你期望的。

3. `now.date()` 是昨天！（或明天）

   如果您始终使用天真的数据时间，那么您可能相信可以通过调用日期时间的 `date()` 方法将日期时间转换为日期。您还认为 `date` 很像 `datetime` ，除非它不太准确。

   在时区感知环境中这不是真的：

```
&gt;&gt;&gt; import datetime
&gt;&gt;&gt; import pytz
&gt;&gt;&gt; paris_tz = pytz.timezone("Europe/Paris")
&gt;&gt;&gt; new_york_tz = pytz.timezone("America/New_York")
&gt;&gt;&gt; paris = paris_tz.localize(datetime.datetime(201
2, 3, 3, 1, 30))
# This is the correct way to convert between time zones with
 pytz.
&gt;&gt;&gt; new_york = new_york_tz.normalize(paris.astimezo
ne(new_york_tz))
&gt;&gt;&gt; paris == new_york, paris.date() == new_york.dat
e()
(True, False)
&gt;&gt;&gt; paris - new_york, paris.date() - new_york.date(
)
(datetime.timedelta(0), datetime.timedelta(1))
&gt;&gt;&gt; paris
datetime.datetime(2012, 3, 3, 1, 30, tzinfo=&lt;DstTzInfo 'E
urope/Paris' CET+1:00:00 STD&gt;)
&gt;&gt;&gt; new_york
datetime.datetime(2012, 3, 2, 19, 30, tzinfo=&lt;DstTzInfo '
America/New_York' EST-1 day, 19:00:00 STD&gt;)
```

   如此示例所示，相同的日期时间具有不同的日期，具体取决于表示时间的时区。但真正的问题是更根本的。

datetime表示时间点。它是绝对的：它不依赖于任何东西。相反，日期是日历概念。这是一段时间，其范围取决于考虑日期的时区。可以看到，这两个概念根本不同，将日期时间转换为日期不是确定性操作。

这在实践中意味着什么？

一般来说，您应避免将 `datetime` 转换为 `date` 。例如，您可以使用 `date` 模板过滤器仅显示日期时间的日期部分。此过滤器将格式化之前将datetime转换为当前时区，以确保结果正确显示。

如果您真的需要自己进行转换，那么必须确保datetime首先转换为适当的时区。通常，这将是当前的时区：

```
&gt;&gt;&gt; from django.utils import timezone
&gt;&gt;&gt; timezone.activate(pytz.timezone("Asia/Singapore
"))
# For this example, we just set the time zone to Singapore,
but here's how
# you would obtain the current time zone in the general case
.
&gt;&gt;&gt; current_tz = timezone.get_current_timezone()
# Again, this is the correct way to convert between time zon
es with pytz.
&gt;&gt;&gt; local = current_tz.normalize(paris.astimezone(c
urrent_tz))
&gt;&gt;&gt; local
datetime.datetime(2012, 3, 3, 8, 30, tzinfo=&lt;DstTzInfo 'A
sia/Singapore' SGT+8:00:00 STD&gt;)
&gt;&gt;&gt; local.date()
datetime.date(2012, 3, 3)
```

4. 我收到错误“ `Are time zone definitions for your database and pytz installe` ” **pytz**安装，所以我想问题是我的数据库？

   如果您使用MySQL，请参阅MySQL注释的*Time zone definitions*部分，了解加载时区定义的说明。

用法

1. 我有一个字符串 `“2012-02-21 10:28:45”` 它位于 `"Europe/Helsinki"` 时区。我如何把它变成一个知道的**datetime**？

   这正是pytz的用途。

```
&gt;&gt;&gt; from django.utils.dateparse import parse_dateti
me
&gt;&gt;&gt; naive = parse_datetime("2012-02-21 10:28:45")
&gt;&gt;&gt; import pytz
&gt;&gt;&gt; pytz.timezone("Europe/Helsinki").localize(naive
, is_dst=None)
datetime.datetime(2012, 2, 21, 10, 28, 45, tzinfo=&lt;DstTzI
nfo 'Europe/Helsinki' EET+2:00:00 STD&gt;)
```

请注意， `localize` 是 `tzinfo` API的pytz扩展。此外，你可能想要捕捉 `pytz.` InvalidTimeError。pytz的文档包含更多示例。您应该在尝试操作感知的数据时间之前查看它。

2. 如何获取当前时区的本地时间？

   嗯，第一个问题是，你真的需要吗？

   当您与人类互动时，您应该只使用本地时间，而且模板图层提供*filters and tags*将数据时间转换为您选择的时区。

   此外，Python知道如何比较感知的数据时间，在必要时考虑UTC偏移。在UTC中编写所有模型和视图代码要容易得多（也可能更快）。因此，在大多数情况下，由 `django.utils.timezone.now()` 返回的UTC中的日期时间就足够了。

   然而，为了完整性，如果你真的想要当前时区的当地时间，你可以如何获得它：

```
&gt;&gt;&gt; from django.utils import timezone
&gt;&gt;&gt; timezone.localtime(timezone.now())
datetime.datetime(2012, 3, 3, 20, 10, 53, 873365, tzinfo=&lt
;DstTzInfo 'Europe/Paris' CET+1:00:00 STD&gt;)
```

   在此示例中，安装pytz，当前时区为 `"Europe/Paris"` 。

3. 如何查看所有可用的时区？

   pytz提供帮助，包括当前时区列表和所有可用时区列表，其中一些只有历史价值。

性能与优化

本文档提供的技术与工具概述，有助于使您的Django代码更高效，更快速，并使用更少系统资源。

简介

通常，人们首先关心的是编写的代码起作用，其逻辑函数根据需要产生预期输出。然而，有时，这将不足以使代码像我们希望的那样有效地工作。

Generally one’s first concern is to write code that works, whose logic functions as required to produce the expected output. Sometimes, however, this will not be enough to make the code work as efficiently as one would like.

General approaches

What are you optimizing *for*?

重要的是有一个清楚的想法你的意思是“性能”。它不只有一个指标。

改进的速度可能是程序最明显的目标，但有时可能会寻求其他性能改进，例如降低内存消耗或减少对数据库或网络的需求。

一个领域的改进往往会带来另一个领域的改进，但并不总是如此；有时甚至可以牺牲另一个。例如，程序速度的提高可能会导致程序使用更多的内存。更糟糕的是，它可以是自我毁灭 - 如果速度的改善是如此内存饥饿，系统开始耗尽内存，你会做更多的危害比好。

还有其他权衡要考虑。你自己的时间是宝贵的资源，比CPU时间更珍贵。一些改进可能太难以实现，或者可能影响代码的可移植性或可维护性。并非所有的性能改进都值得付出努力。

所以，你需要知道你的目标是什么性能改进，你还需要知道你有一个很好的理由，瞄准这个方向，因为你需要：

Performance benchmarking

这是不好的只是猜测或假设在你的代码中低效率。

Django tools

django-debug-toolbar是一个非常方便的工具，可以深入了解代码的工作以及它花费多少时间。特别是它可以显示你的页面生成的所有SQL查询，以及每个人花了多长时间。

第三方面板也可用于工具栏，可以（例如）报告缓存性能和模板呈现时间。

Third-party services

有一些免费服务将从远程HTTP客户端的角度分析和报告您的网站的页面的性能，实际上模拟实际用户的体验。

这些不能报告您的代码的内部，但可以提供有用的洞察您的网站的整体性能，包括不能从Django环境中充分测量的方面。示例包括：

- 雅虎Yslow
- Google PageSpeed

还有一些付费服务执行类似的分析，包括一些是Django感知的，可以与您的代码库集成，以更广泛地分析其性能。

Get things right from the start

一些优化工作涉及解决性能缺陷，但是有些工作可以简单地建立在你所做的工作中，作为你应该采取的良好做法的一部分，甚至在你开始考虑提高性能之前。

在这方面，Python是一个很好的语言，因为看起来优雅和感觉合适的解决方案通常是最好的表现。与大多数技能一样，学习什么“看起来正确”需要练习，但最有用的指南之一是：

Work at the appropriate level

Django提供了许多不同的方法来处理事情，但只是因为它可能以某种方式做某事并不意味着它是最合适的方式。例如，您可能会发现您可以计算同样的事情 - 集合中的项目数量，可能是在 `QuerySet` 中，在Python中，或在模板中。

但是，在较低级别而不是较高级别执行此工作几乎总是更快。在更高层次，系统必须通过多级抽象和多层机械来处理对象。

也就是说，数据库通常可以比Python更快地完成事情，这样做可以比模板语言更快：

```
# QuerySet operation on the database
# fast, because that's what databases are good at
my_bicycles.count()

# counting Python objects
# slower, because it requires a database query anyway, and proce
ssing
# of the Python objects
len(my_bicycles)

# Django template filter
# slower still, because it will have to count them in Python any
way,
# and because of template language overheads
{{ my_bicycles|length }}
```

一般来说，最适合的工作级别是最低级别的代码，它是舒适的。

注意

以上示例仅仅是说明性的。

首先，在现实情况下，您需要考虑在计数之前和之后发生的情况，以确定在特定上下文中执行的最佳方式。数据库优化文档描述了*a case where counting in the template would be better*。

其次，还有其他选择要考虑：在现实生活中，`{{ my_bicycles.count }} t0 >，它直接从模板调用` QuerySet `count()方法可能是最合适的选择。`

Caching

通常，计算值是昂贵的（即资源匮乏和缓慢），因此将值保存到可快速访问的缓存中可以有巨大的好处，为下一次需要做好准备。

这是一个足够重要和强大的技术，Django包括一个综合的缓存框架，以及其他较小的缓存功能。

The caching framework

Django的*caching framework*通过保存动态内容以便不需要为每个请求计算，从而为性能提升提供了非常重要的机会。

为了方便起见，Django提供了不同级别的缓存粒度：您可以缓存特定视图的输出，或仅缓存难以生成的片段，甚至整个网站。

实现缓存不应该被视为改进代码执行不良，因为它已被写得不好的替代方法。这是生产性能良好的代码的最后一步，而不是一个捷径。

`cached_property`

通常必须多次调用类实例的方法。如果这个功能是昂贵的，那么这样做可能是浪费。

使用 `@cached_property` 装饰器保存属性返回的值；下一次在该实例上调用函数时，它将返回保存的值，而不是重新计算它。请注意，这仅适用于将 `self` 作为其唯一参数的方法，并将方法更改为属性。

某些Django组件也有自己的缓存功能；这些在下面在与那些组件相关的部分中讨论。

Understanding laziness

懒惰是对缓存的补充策略。缓存通过保存结果避免重新计算；延迟延迟计算，直到它实际需要。

懒惰允许我们在实例化之前，甚至在可能实例化它们之前引用它们。这有很多用途。

例如，可以在目标语言甚至已知之前使用*lazy translation*，因为在实际需要翻译的字符串之前不发生，例如在呈现的模板中。

懒惰也是一种通过试图避免工作来节省努力的方式。也就是说，懒惰的一个方面是在做任何事情之前不做任何事情，因为它可能不是必然的。因此，懒惰可能具有性能影响，并且相关工作越昂贵，通过懒惰获得的就越多。

Python提供了许多用于延迟评估的工具，特别是通过*generator*和*generator expression*构造。值得一读的是Python中的懒惰，以便发现在代码中使用延迟模式的机会。

Laziness in Django

Django本身相当懒惰。一个很好的例子可以在 `QuerySets` 的评估中找到。*QuerySets are lazy*。因此，可以创建 `QuerySet` ，传递并与其他 `QuerySets` 组合，而不实际引发任何到数据库的访问以获取其描述的项目。传递的是 `QuerySet` 对象，而不是最终需要从数据库中获取的项目集合。

另一方面，*certain operations will force the evaluation of a QuerySet*进行求值。避免对 `QuerySet` 的过早评估可以节省对数据库的昂贵且不必要的访问。

Django还提供了一个 `allow_lazy()` 装饰器。这允许使用延迟参数调用的函数本身行为迟缓，只有在需要时才进行求值。因此，懒惰的论据 - 可能是一个昂贵的论点 - 不会被要求进行评估，直到它是严格要求。

Databases

Database optimization

Django的数据库层提供了各种方法来帮助开发人员从他们的数据库获得最佳性能。*database optimization documentation*汇集了相关文档的链接，并添加了各种提示，概述在尝试优化数据库使用情况时需要采取的步骤。

Other database-related tips

启用*Persistent connections*可以加快连接到数据库帐户的请求处理时间的很大一部分。

这有助于在有限的网络性能的虚拟化主机上很多。

HTTP performance

Middleware

Django附带了一些有用的*middleware*，可以帮助优化您的网站的性能。他们包括：

`ConditionalGetMiddleware`

添加了对现代浏览器的支持，可根据 `ETag` 和 `Last-Modified` 标头有条件地获取响应。

`GZipMiddleware`

压缩所有现代浏览器的响应，节省带宽和传输时间。请注意，GZipMiddleware目前被认为是一种安全风险，并且容易受到由TLS / SSL提供的保护无效的攻击。有关详细信息，请参阅 `GZipMiddleware` 中的警告。

Sessions

Using cached sessions

*Using cached sessions*可能是一种通过消除从像数据库这样较慢的存储源加载会话数据而改为将经常使用的会话数据存储在内存中来提高性能的方法。

Static files

静态文件，根据定义是不动态的，使优化增益的一个优秀的目标。

`CachedStaticFilesStorage`

通过利用Web浏览器的缓存能力，您可以在初始下载后完全消除给定文件的网络匹配。

`CachedStaticFilesStorage` 会将一个内容相关标记附加到*static files*的文件名中，以确保浏览器能够安全地对其进行长期缓存，而不会丢失未来的更改 - 当文件更改时，将标记，所以浏览器将自动重新加载资产。

“Minification”

一些第三方Django工具和包提供了“缩小”HTML，CSS和JavaScript的能力。它们删除不必要的空格，换行符和注释，缩短变量名，从而减少您的网站发布的文档的大小。

Template performance

注意：

- 使用 `{% 阻止 %}` 比使用 `{% 包括 %}`
- 从许多小块组装的重碎片模板可能会影响性能

The cached template loader

启用 `cached template loader` 通常会大幅提高性能，因为它避免每次需要时编译每个模板渲染。

Using different versions of available software

有时可能需要检查您使用的软件的不同和性能更好的版本是否可用。

这些技术针对的是更高级的用户，他们希望提高已经优化的Django站点的性能边界。

然而，他们不是性能问题的魔法解决方案，他们不可能比没有以更正确的方式做更基本的事情的网站带来更好的边际收益。

注意

值得重复的是：达到您已经使用的软件的替代品永远不是性能问题的第一个答案。当您达到这种优化水平时，您需要一个正式的基准解决方案。

Newer is often - but not always - better

很少有新版本的维护良好的软件效率较低，但是维护者无法预期每一种可能的使用情况 - 因此，虽然知道较新的版本可能性能更好，但不要简单地假设它们一直会。

这是Django本身的真实。连续版本已在系统中提供了许多改进，但您仍应检查应用程序的真实性能，因为在某些情况下，您可能会发现这些更改意味着性能更差，而不是更好。

较新的Python版本以及Python软件包，往往会表现得更好 - 但测量，而不是假设。

注意

除非您在特定版本中遇到了异常的性能问题，否则在新版本中通常会找到更好的功能，可靠性和安全性，这些优点远远胜过您可能获胜或失败的任何性能。

Alternatives to Django's template language

对于几乎所有情况，Django的内置模板语言是完全足够的。然而，如果你的Django项目的瓶颈似乎在于模板系统，你已经用尽了其他机会来弥补这一点，第三方替代品可能是答案。

Jinja2可以提高性能，特别是在速度方面。

替代模板系统在它们共享Django的模板语言的程度上有所不同。

注意

如果在模板中遇到性能问题，首先要做的是明白为什么。使用替代模板系统可能会更快，但是也可以获得相同的收益，而不会遇到麻烦 - 例如，您的模板中的昂贵的处理和逻辑可以更有效地在您的视图中。

Alternative software implementations

可能值得检查你使用的Python软件是否已在不同的实现中提供，可以更快地执行相同的代码。

然而：编写良好的Django站点中的大多数性能问题不是在Python执行级别，而是在于低效的数据库查询，缓存和模板。如果你依赖写得不好的Python代码，你的性能问题不可能通过更快地执行来解决。

使用替代实现可能引入兼容性，部署，可移植性或维护问题。不言而喻，在采用非标准实施之前，您应该确保它为您的应用程序提供足够的性能增益，超过潜在的风险。

记住这些注意事项，您应该注意：

PyPy

PyPy是Python中Python本身的实现（'标准'Python实现是在C中）。PyPy可以提供显着的性能提升，通常用于重量级应用。

PyPy项目的一个关键目标是与现有的Python API和库的兼容性。Django是兼容的，但你需要检查你依赖的其他库的兼容性。

C implementations of Python libraries

一些Python库也在C中实现，可以更快。他们的目标是提供相同的API。注意兼容性问题和行为差异是未知的（并不总是立即明显）。

Python 的兼容性

Django 的目的，是要与多个不同的 Python 版本兼容。

Running Django on Jython

Jython 是在 Java 平台 (JVM) 运行的 Python 实现。这个文档将让你在Jython之上运行Django。

Installing Jython

Django使用Jython 2.7b2及更高版本。有关下载和安装说明，请参阅Jython网站。

Creating a servlet container

如果你只是想试验Django，请跳到下一节； Django包含一个可以用于测试的轻量级Web服务器，因此在准备在生产环境中部署Django之前，您不需要设置任何其他内容。

如果要在生产站点上使用Django，请使用Java servlet容器，例如Apache Tomcat。如果您需要其包含的额外功能，则完整的JavaEE应用程序服务器（如GlassFish或JBoss）也可以。

Installing Django

下一步是安装Django本身。这与在标准Python上安装Django完全相同，因此请参阅*Remove any old versions of Django*和*Install the Django code*以获取相关说明。

Installing Jython platform support libraries

django-jython项目包含用于Django / Jython开发的数据库后端和管理命令。注意，内置的Django后端不会在Jython之上工作。

要安装它，请按照项目网站上详细的安装说明操作。另外，阅读数据库后端文档。

Differences with Django on Jython

此时，Jython上的Django应该与运行在标准Python上的Django几乎相同。但是，有几点差异需要牢记：

- 请记住使用 `jython` 命令而不是 `python` 。文档使用 `python` 来保持一致性，但是如果你使用的是Jython，你会想每次发生时用 `jython` 来替代 `python` 。
- 同样，您需要使用 `JYTHONPATH` 环境变量，而不是 `PYTHONPATH` 。
- Django中需要枕头的任何部分都将无法工作。

Porting to Python 3

第一个支持Python 3的版本是Django 1.5。感谢six的兼容层，无需对代码做出任何改动，就可以让你的代码同时在Python 2 (≥ 2.6.5)和Python 3 (≥ 3.2)上运行。

本文档主要针对希望支持Python 2和3的可插拔应用程序的作者。它还描述了适用于Django代码的指南。

Philosophy

本文档假定您熟悉Python 2和Python 3之间的更改。如果你不是，请先阅读Python的官方移植指南。刷新你对Python 2和3的unicode处理的知识将有所帮助；Pragmatic Unicode演示文稿是一个很好的资源。

Django使用*Python 2/3*兼容源策略。当然，你可以为自己的代码自由选择另一个策略，特别是如果你不需要保持与Python 2兼容。但是可插拔应用程序的作者鼓励使用与Django本身相同的移植策略。

如果您定位Python≥2.6，编写兼容的代码要容易得多。Django 1.5引入了诸如 `django.utils.six` 之类的兼容性工具，它是 `six module` 的定制版本， 。为了方便起见，在Django 1.4.2中引入了向前兼容的别名。如果您的应用程序利用这些工具，它将需要Django≥1.4.2。

显然，编写兼容的源代码会增加一些开销，这会导致沮丧。Django的开发人员发现，尝试编写与Python 2兼容的Python 3代码比相反的更有意义。这不仅使你的代码更加面向未来，但Python 3的优势（如saner字符串处理）开始迅速发光。处理Python 2成为向后兼容性的要求，我们作为开发人员来处理这样的约束。

Django提供的移植工具受这种理念的启发，并且贯穿本指南。

Porting tips

Unicode literals

此步骤包括：

- 在Python模块顶部从 t> 未来 导入 unicode_literals添加 `把它放在每个模块中，否则你会继续检查文件的顶部，看看哪个模式是有效的；`
- 在unicode字符串之前删除 `u` 前缀；
- 在bytestrings之前添加 `b` 前缀。

执行这些更改系统地保证向后兼容性。

然而，Django应用程序通常不需要bytestrings，因为Django只将unicode接口暴露给程序员。Python 3不鼓励使用bytestrings，二进制数据或面向字节的接口除外。Python 2使得bytestrings和unicode字符串可以有效地互换，只要它们只包含ASCII数据。利用这一点，尽可能使用unicode字符串，并避免 `b` 前缀。

注意

Python 2的 `u` 前缀是Python 3.2中的一个语法错误，但是由于 **PEP 414**，它将再次允许在Python 3.3中。因此，如果您定位Python≥3.3，则此转换是可选的。它仍然推荐，按照“编写Python 3代码”的哲学。

String handling

Python 2's unicode type was renamed `str` in Python 3, `str()` was renamed `bytes`, and basestring disappeared. 六提供*tools*来处理这些更改。

Django在 `django.utils.encoding` 和 `django.utils.safestring` 模块中还包含多个与字符串相关的类和函数。他们的名字使用了 `str` 这个词，这在Python 2和Python 3以及 `unicode` 中并不是相同的，它在Python 3中不存在。为了避免歧义和混淆，这些概念重命名为 `bytes` 和 `text` 。

以下是 `django.utils.encoding` 中的名称更改：

旧名称	新名称
`smart_str`	`smart_bytes`
`smart_unicode`	`smart_text`
`force_unicode`	`force_text`

为了向后兼容，旧名称仍然适用于Python 2。在Python 3下， `smart_str` 是 `smart_text` 的别名。

为了向前兼容性，新名称的工作原理与Django 1.4.2相同。

注意

`django.utils.encoding` 在Django 1.5中被重构，以提供更一致的API。检查其文档以获取更多信息。

`django.utils.safestring` 主要通过 `mark_safe()` 和 `mark_for_escaping()` 函数使用，但没有更改。如果你使用的内部，这里是名称更改：

旧名称	新名称
`EscapeString`	`EscapeBytes`
`EscapeUnicode`	`EscapeText`
`SafeString`	`SafeBytes`
`SafeUnicode`	`SafeText`

为了向后兼容，旧名称仍然适用于Python 2。在Python 3下，`EscapeString` 和 `SafeString` 分别是 `EscapeText` 和 `SafeText` 的别名。

为了向前兼容性，新名称的工作原理与Django 1.4.2相同。

`__str__()` and unicode() methods

在Python 2中，对象模型指定 `__str__()` 和unicode ()方法。如果这些方法存在，它们必须分别返回 `str` （bytes）和 `unicode` （text）。

`print` 语句和 `str` 内置调用 `__str__()` 来确定对象的人类可读表示。`unicode` 内置调用unicode ()如果存在，否则返回 `__str__()` ，并使用系统编码。相反，`Model` 基类通过编码为UTF-8自动从unicode ()中导出 `__str__()`

在Python 3中，只有 `__str__()` ，它必须返回 `str` （text）。

（也可以定义 `__bytes__()` ，但Django应用程序对该方法几乎没有用处，因为它们几乎不处理 `bytes` 。）

Django provides a simple way to define `__str__()` and unicode() methods that work on Python 2 and 3: you must define a `__str__()` method returning text and to apply the `python_2_unicode_compatible()` decorator.

在Python 3上，装饰器是一个无操作。在Python 2中，它定义了适当的unicode ()和 `__str__()` 方法（替换过程中的原始 `__str__()` 方法）。这里有一个例子：

```
from __future__ import unicode_literals
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class MyClass(object):
    def __str__(self):
        return "Instance of my class"
```

这种技术是Django移植理念的最佳匹配。

对于向前兼容性，此装饰器可用于Django 1.4.2。

最后，请注意， `__repr__()` 必须在所有版本的Python上返回 `str` 。

`dict` and `dict` -like classes

`dict.keys()` ， `dict.items()` 和 `dict.values()` 返回列表在Python 2和迭代器在Python 3。 `QueryDict` 和在 `django.utils.datastructures` 中定义的 `dict`

six provides compatibility functions to work around this change: `iterkeys()` , `iteritems()` , and `itervalues()` . 它还包含未正式记录的 `iterlists` 函数，适用于 `django.utils.datastructures.MultiValueDict` 及其子类。

`HttpRequest` and `HttpResponse` objects

根据 **PEP 3333**：

- 头总是 `str` 对象，
- 输入和输出流始终为 `bytes` 对象。

具体来说， `HttpResponse.content` 包含 `bytes` ，如果您在测试中将其与 `str` 进行比较，则可能会出现问题。首选解决方案是依靠 `assertContains()` 和 `assertNotContains()` 。这些方法接受响应和unicode字符串作为参数。

Coding guidelines

以下准则在Django的源代码中强制实施。它们也推荐用于遵循相同移植策略的第三方应用程序。

Syntax requirements

Unicode

在Python 3中，默认情况下所有字符串都被视为Unicode。来自Python 2的 `unicode` 类型在Python 3中称为 `str` ， `str` 变为 `bytes` 。

您不能在unicode字符串字面量之前使用 `u` 前缀，因为它在Python 3.2中是语法错误。必须使用 `b` 前缀字节字符串。

为了在Python 2中启用相同的行为，每个模块必须从 `__future__` 导入 `unicode_literals` ：

```
from __future__ import unicode_literals

my_string = "This is an unicode literal"
my_bytestring = b"This is a bytestring"
```

如果您需要Python 2下的字节字符串字符串和Python 3下的Unicode字符串字符串，请使用 `str` builtin：

```
str('my string')
```

在Python 3中， `str` 和 `bytes` 之间没有任何自动转换，而且 `codecs` 模块变得更加严格。 `str.encode()` 始终返回 `bytes` ， `bytes.decode` 始终返回 `str` 。因此，有时需要以下模式：

```
value = value.encode('ascii', 'ignore').decode('ascii')
```

如果您必须索引bytestrings，请谨慎。

Exceptions

捕获异常时，使用 `as` 关键字：

```
try:
    ...
except MyException as exc:
    ...
```

这个旧的语法在Python 3中被删除：

```
try:
    ...
except MyException, exc:    # Don't do that!
    ...
```

使用不同跟踪重新处理异常的语法也发生了更改。使用 `six.reraise()` 。

Magic methods

使用下面的模式来处理在Python 3中重命名的魔法方法。

Iterators

```
class MyIterator(six.Iterator):
    def __iter__(self):
        return self             # implement some logic here

    def __next__(self):
        raise StopIteration     # implement some logic here
```

Boolean evaluation

```
class MyBoolean(object):

    def __bool__(self):
        return True             # implement some logic here

    def __nonzero__(self):      # Python 2 compatibility
        return type(self).__bool__(self)
```

Division

```
class MyDivisible(object):

    def __truediv__(self, other):
        return self / other     # implement some logic here

    def __div__(self, other):   # Python 2 compatibility
        return type(self).__truediv__(self, other)

    def __itruediv__(self, other):
        return self // other    # implement some logic here

    def __idiv__(self, other):  # Python 2 compatibility
        return type(self).__itruediv__(self, other)
```

在类上而不是在实例上查找特殊的方法来反映Python解释器的行为。

Writing compatible code with six

six是在单个代码库中支持Python 2和3的规范兼容性库。阅读它的文档！

A `customized version of six` is bundled with Django as of version 1.4.2. 您可以将其导入为 `django.utils.six` 。

以下是编写兼容代码所需的最常见更改。

String handling

在Python 3中删除了 `basestring` 和 `unicode` 类型，并改变了 `str` 的含义。要测试这些类型，请使用以下习语：

```
isinstance(myvalue, six.string_types)       # replacement for basestring
isinstance(myvalue, six.text_type)          # replacement for unicode
isinstance(myvalue, bytes)                  # replacement for str
```

Python≥2.6提供 `bytes` 作为 `str` 的别名，因此您不需要 `six.binary_type` 。

`long`

`long` 类型在Python 3中不再存在。 `1L` 是语法错误。使用 `six.integer_types` 检查值是整数还是长整型：

```
isinstance(myvalue, six.integer_types)      # replacement for (int, long)
```

`xrange`

如果在Python 2上使用 `xrange` ，请导入 `six.moves.range` 并使用它。您还可以导入 `six.moves.xrange` （它等同于 `six.moves.range` ），但第一种方法允许您在删除对Python 2的支持时。

Moved modules

一些模块在Python 3中重命名。 `django.utils.six.moves` 模块（基于 `six.moves module` ）提供了兼容的位置以导入它们。

PY2

如果您需要在Python 2和Python 3中使用不同的代码，请检查 `six.PY2` ：

```
if six.PY2:
    # compatibility code for Python 2
```

这是最后的解决方案，当 `six` 不提供适当的功能。

Django customized version of six

与Django捆绑在一起的六个版本（ `django.utils.six` ）包含一些仅供内部使用的自定义项。

常见的网站应用工具

Django 提供了多种工具用于开发Web应用程序

认证

Django 中的用户认证

Django从开始就带有一个用户认证系统。它处理用户账号、组、权限以及基于cookie的用户会话。本节文档解释默认的实现如何直接使用，以及如何扩展和定制它以适合你项目的需要。

概览

Django认证系统同时处理认证和授权。简单地讲，认证验证一个用户是它们声称的那个人，授权决定一个认证通过的用户允许做什么。这里的词语认证同时指代这两项任务。

认证系统包含：

- 用户
- 权限：二元（是/否）标志指示一个用户是否可以做一个特定的任务。
- 组：对多个用户运用标签和权限的一种通用的方式。
- 一个可配置的密码哈希系统
- 用于登录用户或限制内容的表单和视图
- 一个可插拔的后台系统

Django中的认证系统的目标是非常通用且不提供在web认证系统中某些常见的功能。某些常见问题的解决方法已经在第三方包中实现：

- 密码强度检查
- 登录尝试的制约
- 第三方认证（例如OAuth）

安装

认证的支持作为Django的一个contrib模块，打包于 `django.contrib.auth` 中。默认情况下，要求的配置已经包含在 `django-admin startproject` 生成的 `settings.py` 中，它们的组成包括 `INSTALLED_APPS` 设置中的两个选项：

1. ' `django.contrib.auth` '包含认证框架的核心和默认的模型。
2. ' `django.contrib.contenttypes` '是Django内容类型系统，它允许权限与你创建的模型关联。 和 `MIDDLEWARE_CLASSES` 设置中的两个选项：
3. `SessionMiddleware` 管理请求之间的会话。
4. `AuthenticationMiddleware` 使用会话将用户与请求管理起来。

有了这些设置，运行 `manage.py migrate` 命令将为认证相关的模型创建必要的数据库表并为你的应用中定义的任意模型创建权限。

使用

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

使用Django认证系统

这篇文档解释默认配置下Django认证系统的使用。这些配置已经逐步可以满足大部分常见项目对的需要，可以处理范围非常广泛的任务，且具有一套细致的密码和权限实现。对于需要与默认配置不同需求的项目，Django支持扩展和自定义认证。

Django的认证同时提供认证和授权，并通常统一称为认证系统，因为这些功能某些地方是耦合的。

User对象

`User` 对象是认证系统的核心。它们通常表示与你的站点进行交互的用户，并用于启用限制访问、注册用户信息和关联内容给创建者等。在Django的认证框架中只存在一种类型的用户，因此诸如 `'superusers'` 或管理员 `'staff'` 用户只是具有特殊属性集的user对象，而不是不同类型的user对象。

默认user的基本属性有：

- username
- password
- email
- first_name
- last_name

完整的参考请参阅 `full API documentation` ，该文档更偏重特定的任务。

创建users

创建users最直接的方法是使用 `create_user()` 辅助函数：

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.c
om', 'johnpassword')

# At this point, user is a User object that has already been sav
ed
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.last_name = 'Lennon'
>>> user.save()
```

如果你已经安装了Django admin，你也可以间接地创建*users*.

创建superusers

使用 `createsuperuser` 命令创建superusers：

```
$ python manage.py createsuperuser --username=joe --email=joe@ex
ample.com
```

将会提示你输入一个密码。在你输入一个密码后，该user将会立即创建。如果不带 `--username` 和 `--email` 选项，将会提示你输入这些值。

修改密码

Django不会在user模型上存储原始的（明文）密码，而只是一个哈希（完整的细节参见文档：密码是如何管理的）。因为这个原因，不要尝试直接操作user的password属性。这也是为什么创建一个user时要使用辅助函数。

若要修改一个用户的密码，你有几种选择：

`manage.py changepassword *username*` 提供一种从命令行修改User密码的方法。它提示你修改一个给定user的密码，你必须输入两次。如果它们匹配，新的密码将会立即修改。如果你没有提供user，命令行将尝试修改与当前系统用户匹配的用户名的密码。

你也可以通过程序修改密码，使用 `set_password()` ：

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
>>> u.set_password('new password')
>>> u.save()
```

如果你安装了Django admin，你还可以在认证系统的*admin*页面修改user的密码。

Django还提供*views*和*forms*用于允许user修改他们自己密码。

New in Django 1.7.

如果启用了 `SessionAuthenticationMiddleware` ，修改user的密码将会登出他们所有的会话。 详细信息请参阅密码修改后会话失效。

认证Users

`authenticate` (***credentials*)[source]

认证一个给定用户名和密码，请使用 `authenticate()` 。它以关键字参数形式接收凭证，对于默认的配置它是 `username` 和 `password`，如果密码对于给定的用户名有效它将返回一个 `User` 对象。如果密码无效， `authenticate()` 返回 `None` 。例子：

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # the password verified for the user
    if user.is_active:
        print()
    else:
        print()
else:
    # the authentication system was unable to verify the usernam
e and password
    print()
```

注

这是认证一系列凭证的低级的方法；例如，它被 `RemoteUserMiddleware` 使用。除非你正在编写你自己的认证系统，你可能不会使用到它。当然如果你在寻找一种登录user的方法，请参见 `login_required()` 装饰器。

权限和授权

Django从开始就带有一个简单的权限系统。它提供一种分配权限给特定的用户和用户组的方法。

它被Django的admin站点使用，但欢迎你在你自己的代码中使用。

Django admin 站点使用如下的权限：

- 查看"add"表单并添加一个只限具有该类型对象的“add”权限的用户对象。
- 查看修改列表、查看“change”表单以及修改一个只限具有该类型对象的“change”权限的用户对象。
- 删除一个只限具有该类型对象的“delete”权限的用户对象。

权限不但可以根据每个对象的类型，而且可以根据特定的对象实例设置。通过使用 `ModelAdmin` 类提供的 `has_add_permission()`、`has_change_permission()` 和 `has_delete_perm` 方法，可以针对相同类型的不同对象实例自定义权限。

`User` 对象具有两个多对多的字段：`groups` 和 `user_permissions`。`User` 对象可以用和其它*Django* 模型一样的方式访问它们相关的对象：

```
myuser.groups = [group_list]
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions = [permission_list]
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

默认的权限

当 `django.contrib.auth` 在你的 `INSTALLED_APPS` 设置中列出时，它将确保为你安装的应用中的每个Django模型创建3个默认的权限 – add、change和delete。

这些权限将在你运行 `manage.py migrate` 时创建；在添加 `django.contrib.auth` 到 `INSTALLED_APPS` 中之后，当你第一次运行 `migrate` 时，将会为之前安装的模型创建默认的权限，包括与其同时正在安装的新的模型。之后，每当你运行 `manage.py migrate` 时，它都将为新的模型创建默认的权限。

假设你有个应用的 `app_label` 是 `foo` 和一个名为 `Bar` 的模型，要测试基本的权限，你应该使用：

- add: `user.has_perm('foo.add_bar')`
- change: `user.has_perm('foo.change_bar')`
- delete: `user.has_perm('foo.delete_bar')`

很少直接访问 `Permission` 模型。

组

`django.contrib.auth.models.Group` 模型是用户分类的一种通用的方式，通过这种方式你可以应用权限或其它标签到这些用户。一个用户可以属于任意多个组。

组中某个用户自动具有赋给那个组的权限。例如，如果组 `Site editors` 具有权限 `can_edit_home_page` ，那么该组中的任何用户都具有该权限。

出权限之外，组还是给用户分类的一种方便的方法以给他们某些标签或扩展的功能。例如，你可以创建一个组 `'Special users'` ，然后你可以这样写代码，给他们访问你的站点仅限会员的部分，或者给他们发仅限于会员的邮件。

用程序创建权限

虽然*custom permissions*可以定义在 `Meta` 类中，你还可以直接创建权限。例如，你可以为 `myapp` 中的 `BlogPost` 创建 `can_publish` 权限：

```
from myapp.models import BlogPost
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(BlogPost)
permission = Permission.objects.create(codename='can_publish',
                                       name='Can Publish Posts',
                                       content_type=content_type
)
```

然后该权限可以通过 `user_permissions` 属性分配给一个 `User` ，或者通过 `permissions` 属性分配给 `Group` 。

权限的缓存

`ModelBackend` 在第一次需要访问 `User` 对象来检查权限时会缓存它们的权限。这对于请求-响应循环还是比较好的，因为在权限添加进来之后并不会立即检查（例如在admin中）。如果你正在添加权限并需要立即检查它们，例如在一个测试或视图中，最简单的解决办法是从数据库中重新获取 `User` 。 例如：

```
from django.contrib.auth.models import Permission, User
from django.shortcuts import get_object_or_404

def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    # any permission check will cache the current set of permiss
ions
    user.has_perm('myapp.change_bar')

    permission = Permission.objects.get(codename='change_bar')
    user.user_permissions.add(permission)

    # Checking the cached permission set
    user.has_perm('myapp.change_bar')  # False

    # Request new instance of User
    user = get_object_or_404(User, pk=user_id)

    # Permission cache is repopulated from the database
    user.has_perm('myapp.change_bar')  # True

    ...
```

Web请求中的认证

Django使用会话和中间件来拦截 `request 对象` 到认证系统中。

它们在每个请求上提供一个 `request.user` 属性，表示当前的用户。如果当前的用户没有登入，该属性将设置成 `AnonymousUser` 的一个实例，否则它将是 `User` 的实例。

你可以通过 `is_authenticated()` 区分它们，像这样：

```
if request.user.is_authenticated():
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

如何登入一个用户

如果你有一个认证了的用户，你想把它附带到当前的会话中 - 这可以通过 `login()` 函数完成。

`login` ()[source]

从视图中登入一个用户，请使用 `login()` 。它接受一个 `HttpRequest` 对象和一个 `User` 对象。 `login()` 使用Django的会话框架保存用户的ID在会话中。

注意任何在匿名会话中设置的数据都会在用户登入后的会话中都会记住。

这个例子展示你可能如何使用 `authenticate()` 和 `login()` ：

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
            # Redirect to a success page.
        else:
            # Return a 'disabled account' error message
            ...
    else:
        # Return an 'invalid login' error message.
        ...
```

先调用 `authenticate()` ：

当你是手工登入一个用户时，你必须在调用 `login()` 之前通过 `authenticate()` 成功地认证该用户。 `authenticate()` 在 `User` 上设置一个属性标识哪种认证后台成功认证了该用户（细节参见后台的文档），且该信息在后面登录的过程中是需要的。如果你视图登入一个直接从数据库中取出的用户，将会抛出一个错误。

如何登出一个用户

`logout` ()[source]

若要登出一个已经通过 `django.contrib.auth.login()` 登入的用户，可以在你的视图中使用 `django.contrib.auth.logout()` 。 它接收一个 `HttpRequest` 对象且没有返回值。例如：

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

注意，即使用户没有登入 `logout()` 也不会抛出任何错误。

当你调用 `logout()` 时，当前请求的会话数据将被完全清除。所有存在的数据都将清除。这是为了防止另外一个人使用相同的Web浏览器登入并访问前一个用户的会话数据。如果你想在用户登出之后>可以立即访问放入会话中的数据，请在调用 `django.contrib.auth.logout()` 之后放入。

限制访问给登陆后的用户

原始的方法

限制页面访问的简单、原始的方法是检查 `request.user.is_authenticated()` 并重定向到一个登陆页面：

```
from django.conf import settings
from django.shortcuts import redirect

def my_view(request):
    if not request.user.is_authenticated():
        return redirect('%s?next=%s' % (settings.LOGIN_URL, requ
est.path))
    # ...
```

...或者显示一个错误信息：

```
from django.shortcuts import render

def my_view(request):
    if not request.user.is_authenticated():
        return render(request, 'myapp/login_error.html')
    # ...
```

login_required 装饰器

`login_required` ([*redirect_field_name=REDIRECT_FIELD_NAME*, *login_url=None*])[source]

作为一个快捷方式，你可以使用便捷的 `login_required()` 装饰器：

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

`login_required()` 完成下面的事情：

- 如果用户没有登入，则重定向到settings.LOGIN_URL，并传递当前查询字符串中的绝对路径。例如： `/accounts/login/?next=/polls/3/` 。
- 如果用户已经登入，则正常执行视图。视图的代码可以安全地假设用户已经登入。

默认情况下，在成功认证后用户应该被重定向的路径存储在查询字符串的一个叫做 `)带有一个可选的` redirect_field_name`参数：

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

注意，如果你提供一个值给 `redirect_field_name` ，你非常可能同时需要自定义你的登录模板，因为存储重定向路径的模板上下文变量将使用 `redirect_field_name` 值作为它的键，而不是默认的 `"next"` 。

`login_required()` 还带有一个可选的 `login_url` 参数。例如：

```
from django.contrib.auth.decorators import login_required

@login_required(login_url='/accounts/login/')
def my_view(request):
    ...
```

注意，如果你没有指定 `login_url` 参数，你需要确保 `settings.LOGIN_URL` 并且你的登录视图正确关联。例如，使用默认值，可以添加下面几行到你的URLconf中：

```
from django.contrib.auth import views as auth_views

url(r'^accounts/login/$', auth_views.login),
```

`settings.LOGIN_URL` 同时还接收视图函数名和*命名的URL模式*。这允许你自由地重新映射你的URLconf中的登录视图而不用更新设置。

注

login_required装饰器不检查user的is_active标志位。

给已验证登录的用户添加访问限制

基于特定的权限和其他方式来限制访问，你最好按照前面所叙述的那样操做。

简单的方法就是在视图中直接运行你对 `request.user` 的测试。例如，视图检查用户的邮件属于特定的地址（例如@example.com），若不是，则重定向到登录页面。

```
from django.shortcuts import redirect

def my_view(request):
    if not request.user.email.endswith('@example.com'):
        return redirect('/login/?next=%s' % request.path)
    # ...
```

`user_passes_test` (*func*[, *login_url=None*, *redirect_field_name=REDIRECT_FIELD_NAME*])[source]

你可以用方便的 `user_passes_test` 装饰器，当回掉函数返回 `False` 时会执行一个重定向操作：

```
from django.contrib.auth.decorators import user_passes_test

def email_check(user):
    return user.email.endswith('@example.com')

@user_passes_test(email_check)
def my_view(request):
    ...
```

`user_passes_test()` 要求一个以 `User` 对象为参数的回掉函数，若用户允许访问此视图，返回 `True` 。注意， `user_passes_test()` 不会自动检查 `User` 是否是不是匿名对象。

`user_passes_test()` 接收两个额外的参数：

`login_url`

让你指定那些没有通过检查的用户要重定向至哪里。若不指定其值，它可能是默认的 `settings.LOGIN_URL` 。

`redirect_field_name`

与 `login_required()` 的参数相同。把它设置为 `None` 来把它从 URL 中移除，当你想把通不过检查的用户重定向到没有next page 的非登录页面时。

例如：

```
@user_passes_test(email_check, login_url='/login/')
def my_view(request):
    ...
```

permission_required 装饰器

`permission_required` (*perm*[, *login_url=None*, *raise_exception=False*])[source]

检查一个用户是否有指定的权限是相对常见的需求。因此，Django 提供了一个快捷方式： `permission_required()` 装饰器：

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote')
def my_view(request):
    ...
```

`has_perm()` 方法, 权限名称采用如下方法 `"<app label>.<permission codename>"` (例如 `polls.can_vote` 表示在 `polls` 应用下一个模块的权限。

要注意 `permission_required()` 也接受一个可选的 `login_url` 参数。例如：

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote', login_url='/loginpage/')
def my_view(request):
    ...
```

在 `login_required()` 装饰器中， `login_url` 默认为 `settings.` LOGIN_URL。

如果提供了 `raise_exception` 参数，装饰器抛出 `PermissionDenied` 异常，使用 *the 403 (HTTP Forbidden)* 视图而不是重定向到登录页面。

Changed in Django 1.7:

`permission_required()` 装饰器既可以接收一个权限序列也可以接收一个单个的权限。

对普通的视图使用权限

若要对一个基于类的普通视图使用权限，可以在该类上装饰 `View.dispatch` 方法。详细细节参见*Decorating the class*。 另外一个方法是编写一个封装*as_view()*的*mixin*。

密码更改后的会话失效

New in Django 1.7.

警告

这种保护只在 `MIDDLEWARE_CLASSES` 中 `SessionAuthenticationMiddleware` 开启的情况下应用。如果 `settings.py` 由Django ≥ 1.7. 的 `startproject` 生成，它会被包含进来。

在Django 2.0中，会话验证会变成强制性的， 无论是否开启了 `SessionAuthenticationMiddleware` 。 如果你拥有一个1.7之前的项目，或者使用不包含 `SessionAuthenticationMiddleware` 的模板生成的项目，考虑在阅读下面的升级说明之后开启它。

如果你的 `AUTH_USER_MODEL` 继承自 `AbstractBaseUser` ，或者实现了它自己的 `get_session_auth_hash()` 方法，验证后的会话会包含这个函数返回的哈希值。在 `AbstractBaseUser` 的情况中，这是密码字段的HMAC。如果开启了 `SessionAuthenticationMiddleware` ，Django会验证每个请求带有的哈希值是否匹配服务端计算出来的哈希值。这允许用户通过修改密码来登出所有的会话。

Django中包含的默认的密码修改视图，以及 `django.contrib.auth` 中的 `django.contrib.auth.views.password_change()` 和 `user_change_passwo` 视图 ，会使用新的密码哈希值升级会话，以便用户在修改密码是不会登出。如果你拥有自定义的密码修改视图，并且希望具有相似的行为，使用这个函数：

`update_session_auth_hash` (*request*, *user*)

这个函数接受当前请求，并且会在会话哈希值得到的地方升级用户对象，也会适当地升级会话哈希值。使用示例：

```
from django.contrib.auth import update_session_auth_hash

def password_change(request):
    if request.method == 'POST':
        form = PasswordChangeForm(user=request.user, data=reques
t.POST)
        if form.is_valid():
            form.save()
            update_session_auth_hash(request, form.user)
    else:
        ...
```

如果你在升级一个现存的站点，并且希望开启这一中间件，而不希望你的所有用户之后重新登录，你可以首先升级到DJango1.7并且运行它一段时间，以便所有会话在用户登录时自然被创建，它们包含上面描述的会话哈希。一旦你使用 `SessionAuthenticationMiddleware` 开始运行你的站点，任何没有登录并且会话使用验证哈希值升级过的用户的现有会话都会失效，并且需要重新登录。

注意

虽然 `get_session_auth_hash()` 给予 `SECRET_KEY` ，使用新的私钥升级你的站点会使所有现有会话失效。

认证的视图

Django提供一些视图，你可以用来处理登录、登出和密码管理。它们使用*stock auth* 表单，但你也可以传递你自己的表单。

Django没有为认证视图提供默认的模板。你应该为你想要使用的视图创建自己的模板。模板的上下文定义在每个视图中，参见所有的认证视图.

使用视图

有几种不同的方法在你的项目中使用这些视图。最简单的方法是包含 `django.contrib.auth.urls` 中提供的URLconf到你自己的URLconf中，例如

```
urlpatterns = [
    url('^', include('django.contrib.auth.urls'))
]
```

这将包含进下面的URL模式：

```
^login/$ [name='login']
^logout/$ [name='logout']
^password_change/$ [name='password_change']
^password_change/done/$ [name='password_change_done']
^password_reset/$ [name='password_reset']
^password_reset/done/$ [name='password_reset_done']
^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[
0-9A-Za-z]{1,20})/$ [name='password_reset_confirm']
^reset/done/$ [name='password_reset_complete']
```

这些视图提供了一个简单易记的URL名称。使用命名URL模式的细节请参见*URL*文档。

如果你想更多地控制你的URL，你可以在你的URLconf中引用一个特定的视图：

```
urlpatterns = [
    url('^change-password/', 'django.contrib.auth.views.password
_change')
]
```

这些视图具有可选的参数，你可以用来改变视图的行为。例如，如果你想修改一个视图使用的模板名称，你可以提供 `template_name` 参数。实现它的一种方法是在URLconf中提供一个关键字参数，它们将被传递到视图中。例如：

```
urlpatterns = [
    url(
        '^change-password/',
        'django.contrib.auth.views.password_change',
        {'template_name': 'change-password.html'}
    )
]
```

所有的视图都返回一个 `TemplateResponse` 实例，这允许你在渲染之前很容易自定义响应。实现它的一种方法是在你自己的视图中包装一个视图：

```
from django.contrib.auth import views

def change_password(request):
    template_response = views.password_change(request)
    # Do something with `template_response`
    return template_response
```

更多的细节，参见*TemplateResponse*文档。

所有的认证视图

下面列出了 `django.contrib.auth` 提供的所有视图。实现细节参见使用视图。

`login` (*request*[, *template_name*, *redirect_field_name*, *authentication_form*, *current_app*, *extra_context*])[source]

URL 名称： `login`

关于使用命名URL模式的细节参见*URL* 文档。

可选的参数：

- `template_name` : 用于用户登录视图的模板名。默认为 `registration/login.html` 。
- `redirect_field_name` : `GET` 字段的名称，包含登陆后重定向URL。默认为 `next` 。
- `authentication_form` : 用于认证的可调用对象（通常只是一个表单类）。默认为AuthenticationForm。
- `current_app` : 指示包含当前视图的是哪个应用。更多信息参见命名URL的解析策略。
- `extra_context` : 一个上下文数据的字典，将被添加到传递给模板的默认上下文数据中。

下面是 `django.contrib.auth.views.login` 所做的事情：

- 如果通过 `GET` 调用，它显示一个POST给相同URL的登录表单。后面有更多这方面的信息。
- 如果通过 `POST` 调用并带有用户提交的凭证，它会尝试登入该用户。如果登入成功，该视图重定向到 `next` 中指定的URL。如果 `next` 没有提供，它重定向到settings.LOGIN_REDIRECT_URL（默认为 `/accounts/profile/` ）。如果登入不成功，则重新显示登录表单。

你需要提供html模板给login，默认调用 `registration/login.html` 。模板会得到4个模板上下文变量：

- `form` : 一个表示AuthenticationForm的Form对象。
- `next` : 登入成功之后重定向的URL。它还可能包含一个查询字符串。
- `site` : 如果你没有安装site框架，这将被设置成RequestSite的一个实例，它

从当前的HttpRequest获得site名称和域名。

- `site_name` : `site.name` 的别名。如果你没有安装site框架，这将被设置成request.META['SERVER_NAME']的值。关于site 的更多信息，参见*"sites"* 框架。

如果你不喜欢调用 `registration/login.html` ，你可以通过额外的参数传递 `template_name` 参数给你URLconf中的视图。例如，下面URLconf中的行将使用 `myapp/login.html` ：

```
url(r'^accounts/login/$', auth_views.login, {'template_name': 'myapp/login.html'}),
```

通过传递 `redirect_field_name` 给视图，你还可以指定 `GET` 字段的值，它包含登入成功后的重定向的URL。默认情况下，该字段叫做 `next` 。

下面是一个 `registration/login.html` 模板的示例，你可以用它来作为起点。它假设你有一个定义了 `content` 块的 `base.html` 模板：

```
{% extends "base.html" %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

如果你自定义认证（参见*Customizing Authentication*），你可以通过 `authentication_form` 参数传递一个自定义的认证表单给登录视图。该表单必须在它的 `__init__` 方法中接收一个 `request` 关键字参数，并提供一个 `get_user` 方法，此方法返回认证过的用户对象（这个方法永远只在表单验证成功后调用）。

`logout` (*request*[, *next_page*, *template_name*, *redirect_field_name*, *current_app*, *extra_context*])[source]

登出一个用户。

URL名称： `logout`

可选的参数：

- `next_page`：登出之后要重定向的URL。
- `template_name`：用户登出之后，要展示的模板的完整名称。如果不提供任何参数，默认为 `registration/logged_out.html` 。
- `redirect_field_name`：包含登出之后所重定向的URL的 `GET` 字段的名称。默认为 `next` 。如果提供了 `GET` 参数，会覆盖 `next_page` URL。
- `current_app`：一个提示，表明哪个应用含有了当前视图。 详见 命名空间下的URL解析策略 。
- `extra_context`：一个上下文数据的字典，会被添加到向模板传递的默认的上下文数据中。

模板上下文：

- `title`：本地化的字符串"登出"。
- `site`：根据SITE_ID 设置的当前站点。如果你并没有安装站点框架，会设置为 RequestSite的示例，它从当前HttpRequest来获取站点名称和域名。
- `site_name`： `site.name` 的别名。如果没有安装站点框架，会设置为request.META['SERVER_NAME']。站点的更多信息请见"站点"框架。
- `current_app`：一个提示，表明哪个应用含有了当前视图。 详见 命名空间下的URL解析策略 。
- `extra_context`：一个上下文数据的字典，会被添加到向模板传递的默认的上下文数据中。

`logout_then_login` (*request*[, *login_url*, *current_app*, *extra_context*])[source]

登出一个用户，然后重定向到登录页面。

URL 名称： 没有提供默认的URL

可选的参数：

- `login_url`：登录页面要重定向的URL。如果没有提供，默认为settings.LOGIN_URL。
- `current_app`：一个提示，表明哪个应用含有了当前视图。详见 命名空间下的URL解析策略 。
- `extra_context`：一个上下文数据的字典，会被添加到向模板传递的默认的上下文数据中。

`password_change` (*request*[, *template_name*, *post_change_redirect*, *password_change_form*, *current_app*, *extra_context*])[source]

允许一个用户修改他的密码。

URL 名称： `password_change`

可选的参数：

- `template_name`：用来显示修改密码表单的template的全名。如果没有提供，默认为 `registration/password_change_form.html` 。
- `post_change_redirect`：密码修改成功后重定向的URL。
- `password_change_form`：一个自定义的"修改密码"表单，必须接受 `user` 关键词参数。表单用于实际修改用户密码。默认为 PasswordChangeForm。
- `current_app`：一个提示，暗示哪个应用包含当前的视图。详见 命名空间下的URL解析策略 。
- `extra_context`：上下文数据的字典，会添加到传递给模板的默认的上下文数据中。

模板上下文：

- `form`：密码修改表单（请见上面的 `password_change_form` ）。

`password_change_done` (*request*[, *template_name*, *current_app*, *extra_context*])[source]

这个页面在用户修改密码之后显示。

URL 名称： `password_change_done`

可选参数：

- `template_name`：所使用模板的完整名称。如果没有提供，默认为 `registration/password_change_done.html` 。
- `current_app`：一个提示，暗示哪个应用包含当前的视图。 详见 命名空间下的URL解析策略 。
- `extra_context`：上下文数据的字典，会添加到传递给模板的默认的上下文数据中。

`password_reset` (*request*[, *is_admin_site*, *template_name*, *email_template_name*, *password_reset_form*, *token_generator*, *post_reset_redirect*, *from_email*, *current_app*, *extra_context*, *html_email_template_name*])[source]

允许用户通过生成一次性的连接并发送到用户注册的邮箱地址中来重置密码。

如果提供的邮箱地址不在系统中存在，这个视图不会发送任何邮件，但是用户也不会收到任何错误信息。这会阻止数据泄露给潜在的攻击者。如果你打算在这种情况提供错误信息，你可以继承 `PasswordResetForm` ，并使用 `password_reset_form` 参数。

用无效密码标记的用户（参见 `set_unusable_password()` ）不允许请求重置密码，为了防止使用类似于LDAP的外部验证资源时的滥用。注意它们不会收到任何错误信息，因为这会暴露它们的账户，也不会发送任何邮件。

URL 名称： `password_reset`

可选参数：

- `template_name` : The full name of a template to use for displaying the password reset form. Defaults to `registration/password_reset_form.html` if not supplied.
- `email_template_name` : The full name of a template to use for generating the email with the reset password link. Defaults to `registration/password_reset_email.html` if not supplied.
- `subject_template_name` : The full name of a template to use for the subject of the email with the reset password link. Defaults to `registration/password_reset_subject.txt` if not supplied.
- `password_reset_form` : Form that will be used to get the email of the user to reset the password for. Defaults to {{s.379}}.
- `token_generator` : Instance of the class to check the one time link. This will default to `default_token_generator` , it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator` .
- `post_reset_redirect` : The URL to redirect to after a successful password reset request.
- `from_email` : A valid email address. By default Django uses the DEFAULT_FROM_EMAIL.
- `current_app` : A hint indicating which application contains the current view. See the {{s.385}} for more information.
- `extra_context` : A dictionary of context data that will be added to the default context data passed to the template.
- `html_email_template_name` : The full name of a template to use for generating a `text/html` multipart email with the password reset link. By default, HTML email is not sent.

New in Django 1.7:

添加了 `html_email_template_name` 。

Deprecated since version 1.8: `is_admin_site` 参数已被废弃，将在Django2.0中被移除。

模板上下文：

- `form` : The form (see `password_reset_form` above) for resetting the user's password.

Email模板上下文：

- `email` : An alias for `user.email`
- `user` : The current User, according to the `email` form field. Only active

users are able to reset their passwords (`User.is_active is True`).
- `site_name` : An alias for `site.name` . If you don't have the site framework installed, this will be set to the value of request.META['SERVER_NAME']. For more on sites, see *The "sites" framework*.
- `domain` : An alias for `site.domain` . If you don't have the site framework installed, this will be set to the value of `request.get_host()` .
- `protocol` : http or https
- `uid` : The user's primary key encoded in base 64.
- `token` : Token to check that the reset link is valid.

`registration/password_reset_email.html` 样例（邮件正文模板）：

```
Someone asked for password reset for email {{ email }}. Follow t
he link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb
64=uid token=token %}
```

主题模板使用了同样的模板上下文。主题必须是单行的纯文本字符串。

`password_reset_done` (*request*[, *template_name*, *current_app*, *extra_context*]) [source]

这个页面在向用户发送重置密码的邮件后展示。如果 `password_reset()` 视图没有显式设置 `post_reset_redirect` URL，默认会调用这个视图。

URL名称： `password_reset_done`

注意

如果提供的email地址在系统中不存在，用户未激活，或者密码不可用，用户仍然会重定向到这个视图，但是不会发送邮件。

可选参数：

- `template_name` : The full name of a template to use. Defaults to `registration/password_reset_done.html` if not supplied.
- `current_app` : A hint indicating which application contains the current view. See the {{s.393}} for more information.
- `extra_context` : A dictionary of context data that will be added to the default context data passed to the template.

`password_reset_confirm` (*request*[, *uidb64*, *token*, *template_name*, *token_generator*, *set_password_form*, *post_reset_redirect*, *current_app*, *extra_context*])[source]

为输入新密码展示表单。

URL名称： `password_reset_confirm`

可选参数：

- `uidb64` : The user's id encoded in base 64. Defaults to `None` .
- `token` : Token to check that the password is valid. Defaults to `None` .
- `template_name` : The full name of a template to display the confirm password view. Default value is `registration/password_reset_confirm.html` .
- `token_generator` : Instance of the class to check the password. This will default to `default_token_generator` , it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator` .
- `set_password_form` : Form that will be used to set the password. Defaults to {{s.395}}
- `post_reset_redirect` : URL to redirect after the password reset done. Defaults to `None` .
- `current_app` : A hint indicating which application contains the current view. See the {{s.400}} for more information.
- `extra_context` : A dictionary of context data that will be added to the default context data passed to the template.

Template context:

- `form` : The form (see `set_password_form` above) for setting the new user's password.
- `validlink` : Boolean, True if the link (combination of `uidb64` and `token`) is valid or unused yet.

`password_reset_complete` (*request*[, *template_name*, *current_app*, *extra_context*])[source]

展示一个视图，它通知用户密码修改成功。

URL名称： `password_reset_complete`

可选参数：

- `template_name` : The full name of a template to display the view. Defaults to `registration/password_reset_complete.html` .
- `current_app` : A hint indicating which application contains the current view. See the {{s.403}} for more information.
- `extra_context` : A dictionary of context data that will be added to the default context data passed to the template.

辅助函数

`redirect_to_login` (*next*[, *login_url*, *redirect_field_name*])[source]

重定向到登录页面，然后在登入成功后回到另一个URL。

必需的参数：

- `next` : The URL to redirect to after a successful login.

可选的参数：

- `login_url` : The URL of the login page to redirect to. Defaults to {{s.411}} if not supplied.
- `redirect_field_name` : The name of a `GET` field containing the URL to redirect to after log out. Overrides `next` if the given `GET` parameter is passed.

内建的表单

如果你不想用内建的视图，但是又不想编写针对该功能的表单，认证系统提供了几个内建的表单，位于 `django.contrib.auth.forms` ：

注

内建的验证表单对他们处理的用户模型做了特定假设。如果你使用了自定义的用户模型，可能需要为验证系统定义你自己的表单。更多信息请见 使用带有自定义用户模型的内建验证表单的文档。

class `AdminPasswordChangeForm` [source]

管理界面中使用的表单，用于修改用户密码。

接受 `user` 作为第一个参数。

class `AuthenticationForm` [source]

用于用户登录的表单。

接受 `request` 作为第一个参数，它储存在表单实例中，被子类使用。

`confirm_login_allowed` (*user*)[source]

New in Django 1.7.

通常， `AuthenticationForm` 会拒绝 `is_active` 标志是 `False` 的用户。你可以使用自定义政策覆盖这一行为，来决定哪些用户可以登录。使用一个继承 `AuthenticationForm` 并覆写 `confirm_login_allowed` 方法的自定义表单来实现它。如果提供的用户不能登录，这个方法应该抛出 `ValidationError` 异常。

例如，允许所有用户登录，不管“活动”状态如何：

```
from django.contrib.auth.forms import AuthenticationForm

class AuthenticationFormWithInactiveUsersOkay(AuthenticationForm
):
    def confirm_login_allowed(self, user):
        pass
```

或者只允许一些活动用户登录进来：

```
class PickyAuthenticationForm(AuthenticationForm):
    def confirm_login_allowed(self, user):
        if not user.is_active:
            raise forms.ValidationError(
                _(),
                code='inactive',
            )
        if user.username.startswith('b'):
            raise forms.ValidationError(
                _(),
                code='no_b_users',
            )
```

class `PasswordChangeForm` [source]

一个表单，允许用户修改他们的密码。

class `PasswordResetForm` [source]

一个表单，用于生成和通过邮件发送一次性密码重置链接。

`send_email` (*subject_template_name*, *email_template_name*, *context*, *from_email*, *to_email*[, *html_email_template_name=None*])

New in Django 1.8.

使用参数来发送 `EmailMultiAlternatives` 。可以覆盖来自定义邮件如何发送给用户。

Parameters:

- **subject_template_name** – the template for the subject.
- **email_template_name** – the template for the email body.
- **context** – context passed to the `subject_template` , `email_template` , and `html_email_template` (if it is not `None`).
- **from_email** – the sender's email.
- **to_email** – the email of the requester.
- **html_email_template_name** – the template for the HTML body; defaults to `None` , in which case a plain text email is sent.

通常， `save()` 位于 `context` 中，并带有 `password_reset()` 向它的email上下文传递的一些变量。

class `SetPasswordForm` [source]

允许用户不输入旧密码修改密码的表单。

class `UserChangeForm` [source]

用户管理界面中修改用户信息和许可的表单。

class `UserCreationForm` [source]

用于创建新用户的表单。

模板中的认证数据

当你使用 `RequestContext` 时，当前登入的用户和它们的权限在模板上下文中可以访问。

技术细节

技术上讲，这些变量只有在你使用 `RequestContext` 并启用了 `'django.contrib.auth.context_processors.auth'` 上下文处理器时才可以在模板上下文中访问到。它是默认产生的配置文件。更多信息，参见 *RequestContext 文档*。

用户

当渲染 `RequestContext` 模板时，当前登录的用户，可能是 `User` 实例或者 `AnonymousUser` 实例，会存储在模板变量 `{{ user }}` 中：

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

如果使用的不是 `RequestContext` ，则不可以访问该模板变量：

权限

当前登录的用户的权限存储在模板变量 `{{ perms }}` 中。这是个 `django.contrib.auth.context_processors` 实例的封装，他是一个对于模板友好的权限代理。

在 `{{ perms }}` 对象中，单一属性的查找是 `User.has_module_perms` 的代理。如果已登录的用户在 `foo` 应用中拥有任何许可，这个例子会显示 `True` ：

```
{{ perms.foo }}
```

二级属性的查找是 `User.has_perm` 的代理。如果已登录的用户拥有 `foo.can_vote` 的许可，这个示例会显示 `True` ：

```
{{ perms.foo.can_vote }}
```

所以，你可以用模板的 `{% if %}` 语句检查权限：

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.can_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.<
/p>
{% endif %}
```

还可以通过 `{% if in %}` 语句查询权限。例如：

```
{% if 'foo' in perms %}
    {% if 'foo.can_vote' in perms %}
        <p>In lookup works, too.</p>
    {% endif %}
{% endif %}
```

在admin中管理用户

如果 `django.contrib.admin` 和 `django.contrib.auth` 这两个你都安装了，将可以通过admin方便地查看和管理用户、组和权限。可以像其它任何Django模型一样创建和删除用户。可以创建组，并分配权限给用户和组。admin中还会保存和显示对用户模型编辑的日志。

创建用户

在admin的主页，你应该可以在“Auth”部分看到“Users”链接。“Add user” 页面与标准admin页面不同点在于它要求你在编辑用户的其它字段之前先选择一个用户名和密码。

另请注意：如果你想使得一个用户能够使用Django的admin站点创建其它用户， 你需要给他添加用户和修改用户的权限（例如，"Add user” 和“Change user” 权限）。如果一个账号具有添加用户的权限但是没有权限修改他们，该账号将不能添加用户。为什么呢？因为如果你具有添加用户的权限，你将可以添加超级用户，这些超级用户将可以修改其他用户。所以Django同时要求添加权限和修改权限作为一种轻量的安全措施。

仔细考虑一下你是如何允许用户管理权限的。如果你了一个非超级用户编辑用户的能力，这和给他们超级用户的权限在最终效果上是一样的，因为他们将能够提升他们自己下面的用户的权限。

修改密码

用户密码不会显示在admin上（也不会存储在数据库中），但是会显示 密码存储的细节。 这个信息的显示中包含一条指向修改密码表单的链接，允许管理员修改用户的密码。

> 译者：Django 文档协作翻译小组，原文：Using the authentication system 。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Django中的密码管理

密码管理在非必要情况下一般不会重新发明，Django致力于提供一套安全、灵活的工具集来管理用户密码。本文档描述Django存储密码和hash存储方法配置的方式，以及使用hash密码的一些实例。

另见

即使用户可能会使用强密码，攻击者也可能窃听到他们的连接。使用*HTTPS*来避免在HTTP连接上发送密码（或者任何敏感的数据），因为否则密码又被嗅探的风险。

Django如何储存密码

Django通常使用PBKDF2来提供灵活的密码储存系统。

`User` 对象的 `password` 属性是一个这种格式的字符串：

```
<algorithm>$<iterations>$<salt>$<hash>
```

那些就是用于储存用户密码的部分，以美元字符分分隔。它们由哈希算法、算法迭代次数（工作因数）、随机的salt、以及生成的密码哈希值组成。算法是Django可以使用的，单向哈希或者密码储存算法之一，请见下文。迭代描述了算法在哈希上执行的次数。salt是随机的种子值，哈希值是这个单向函数的结果。

通常，Django以SHA256的哈希值使用PBKDF2算法，由NIST推荐的一种密码伸缩机制。这对于大多数用户都很有效：它非常安全，需要大量的计算来破解。

然而，取决于你的需求，你可以选择一个不同的算法，或者甚至使用自定义的算法来满足你的特定的安全环境。不过，大多数用户并不需要这样做 -- 如果你不确定，最好不要这样。如果你打算这样做，请继续阅读：

DJango通过访问 `PASSWORD_HASHERS` 设置来选择要使用的算法。这里有一个列表，列出了Django支持的哈希算法类。列表的第一个元素(即 `settings.` PASSWORD_HASHERS[0]) 会用于储存密码， 所有其它元素都是用于验证的哈希值，它们可以用于检查现有的密码。意思是如果你打算使用不同的算法，你需要修改 `PASSWORD_HASHERS` ，来将你最喜欢的算法在列表中放在首位。

`PASSWORD_HASHERS` 默认为：

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
)
```

这意味着，Django会使用 PBKDF2 储存所有密码，但是支持使用 PBKDF2SHA1, bcrypt, SHA1等等算法来检查储存的密码。下一节会描述一些通用的方法，高级用户可能想通过它来修改这个设置。

在Django中使用bcrypt

Bcrypt是一种流行的密码储存算法，它特意被设计用于长期的密码储存。Django并没有默认使用它，由于它需要使用三方的库，但是由于很多人都想使用它，Django会以最小的努力来支持。

执行以下步骤来作为你的默认储存算法来使用Bcrypt：

1. 安装bcrypt 库。这可以通过运行 `pip install django[bcrypt]`，或者下载并运行 `python setup.py install` 来实现。
2. 修改 `PASSWORD_HASHERS` ，将 `BCryptSHA256PasswordHasher` 放在首位。也就是说，在你的设置文件中应该：

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher'
,
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
)
```

(你应该将其它元素留在列表中，否则Django不能升级密码；见下文)。

配置完毕 -- 现在Django会使用Bcrypt作为默认的储存算法。

BCryptPasswordHasher的密码截断

bcrypt的设计者会在72个字符处截断所有的密码，这意味着 `bcrypt(password_with_100_chars) == bcrypt(password_with_100_chars|` 。原生的 `BCryptPasswordHasher` 并不会做任何的特殊处理， 所以它也会受到

这一隐藏密码长度限制的约束。 `BCryptSHA256PasswordHasher` 通过事先使用 sha256生成哈希来解决这一问题。这样就可以防止密码截断了，所以你还是应该优先考虑 `BCryptPasswordHasher` 。这个截断带来的实际效果很微不足道，因为大多数用户不会使用长度超过72的密码，并且即使在72个字符处截断，破解brypt所需的计算能力依然是天文数字。虽然如此，我们还是推荐使用 `BCryptSHA256PasswordHasher` ，根据“有备无患”的原则。

其它 bcrypt 的实现

有一些其它的bcrypt 实现，可以让你在Django中使用它。Django的bcrypt 支持并不直接兼容这些实现。你需要修改数据库中的哈希值，改为 `bcrypt$(raw bcrypt output)` 的形式，来升级它们。例如： `bcrypt$$2a$12$NT0I31Sa7ihGEWpka9ASYrEFkhuTNeBQ2xfZskIiiJeyFXhRgS.Sy` 。

增加工作因数

PBKDF2 和bcrypt 算法使用大量的哈希迭代或循环。这会有意拖慢攻击者，使对哈希密码的攻击更难以进行。然而，随着计算机能力的不断增加，迭代的次数也需要增加。我们选了一个合理的默认值（并且在Django的每个发行版会不断增加），但是你可能想要调高或者调低它，取决于你的安全需求和计算能力。要想这样做，你可以继承相应的算法，并且覆写 `iterations` 参数。例如，增加PBKDF2算法默认使用的迭代次数：

1. 创建 `django.contrib.auth.hashers.PBKDF2PasswordHasher` 的子类：

   ```
   from django.contrib.auth.hashers import PBKDF2PasswordHasher

   class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
       """
       A subclass of PBKDF2PasswordHasher that uses 100 times m
   ore iterations.
       """
       iterations = PBKDF2PasswordHasher.iterations * 100
   ```

   把它保存在项目中的某个位置。例如，把它放在类似于 `myproject/hashers.py` 的文件中。

2. 将你的新的hasher作为第一个元素添加到 `PASSWORD_HASHERS` ：

```
PASSWORD_HASHERS = (
    'myproject.hashers.MyPBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher'
,
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
)
```

配置完毕 -- 现在DJango在储存使用PBKDF2的密码时会使用更多的迭代次数。

密码升级

用户登录之后，如果他们的密码没有以首选的密码算法来储存，Django会自动将算法升级为首选的那个。这意味着Django中旧的安装会在用户登录时自动变得更加安全，并且你可以随意在新的（或者更好的）储存算法发明之后切换到它们。

然而，Django只会升级在 `PASSWORD_HASHERS` 中出现的算法，所以升级到新系统时，你应该确保不要 移除列表中的元素。如果你移除了，使用列表中没有的算法的用户不会被升级。修改PBKDF2迭代次数之后，密码也会被升级。

Manually managing a user’s password

`django.contrib.auth.hashers` 模块提供了一系列的函数来创建和验证哈希密码。你可以独立于 `User` 模型之外使用它们。

`check_password` (*password*, *encoded*)[source]

如果你打算通过比较纯文本密码和数据库中哈希后的密码来手动验证用户，要使用 `check_password()` 这一便捷的函数。它接收两个参数：要检查的纯文本密码，和数据库中用户的 `password` 字段的完整值。如果二者匹配，返回 `True` ，否则返回 `False` 。

`make_password` (*password*, *salt=None*, *hasher='default'*)[source]

以当前应用所使用的格式创建哈希密码。它接受一个必需参数：纯文本密码。如果你不想使用默认值（ `PASSWORD_HASHERS` 设置的首选项），你可以提供salt值和要使用的哈希算法，它们是可选的。当前支持的算法是： `'pbkdf2_sha256'` , `'pbkdf2_sha1'` , `'bcrypt_sha256'` (参见*在 Django中使用Bcrypt*), `'bcrypt'` , `'sha1'` , `'md5'` , `'unsalted_md5'` (仅仅用于向后兼容) 和 `'crypt'` （如果你安装了 `crypt` 库）。如果password参数是 `None` ，会返回一个不可用的密码（它永远不会被 `check_password()` 接受）。

`is_password_usable` (*encoded_password*)[source]

检查提供的字符串是否是可以用 `check_password()` 验证的哈希密码。

> 译者：Django 文档协作翻译小组，原文：Password management。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

在Django中自定义身份验证

Django 自带的认证系统足够应付大多数情况，但你或许不打算使用现成的认证系统。定制自己的项目的权限系统需要了解哪些一些关键点，即Django中哪些部分是能够扩展或替换的。这个文档提供了如何定制权限系统的细节。

“认证”后端 在以下情形时可被扩展:当一个 User 模型对象带有用户名和密码时，且需要有别于 Django 默认的认证功能。

你可为你的模型提供可通过 Django 权限系统检查的 定制的权限。

你能够扩展 默认的 User 模型，或实现 一个完全定制的模型。

其他认证源

有时候你需要挂接到其他认证资源 -- 另一包含用户名，密码的数据源或者其他认证方法。

例如，您的公司可能已经有一个LDAP设置存储了每一位员工的用户名和密码。 对于一个在LDAP和Django网站都拥有账号的用户来说，如果他/她不能使用LDAP账号登录Django网站，对他/她以及网站管理员来说都是一件麻烦事。

为了解决类似情形，django的认证系统允许你添加其他认证方式。您可以覆盖Django的基于数据库的默认方案，也可以连接使用其他系统的认证服务。

有关Django附带的身份验证后端的信息，请参阅*authentication backend reference*。

指定认证后端

在底层，Django 维护一个“认证后台”的列表。当调用 `django.contrib.auth.authenticate()` 时 —— 如何登入一个用户 中所描述的 —— Django 会尝试所有的认证后台进行认证。如果第一个认证方法失败，Django 将尝试第二个，以此类推，直至试完所有的认证后台。

使用的认证后台通过 `AUTHENTICATION_BACKENDS` 设置指定。它应该是一个包含Python 路径名称的元组，它们指向的Python 类知道如何进行验证。这些类可以位于Python 路径上任何地方。

默认情况下， `AUTHENTICATION_BACKENDS` 设置为：

```
('django.contrib.auth.backends.ModelBackend',)
```

这个基本的认证后台会检查Django 的用户数据库并查询内建的权限。它不会通过任何的速率限制机制防护暴力破解。你可以在自定义的认证后端中实现自己的速率控制机制，或者使用大部分Web 服务器提供的机制。

`AUTHENTICATION_BACKENDS` 的顺序很重要，所以如果用户名和密码在多个后台中都是合法的，Django 将在第一个匹配成功后停止处理。

如果后台引发 `PermissionDenied` 异常，认证将立即失败。Django 不会检查后面的认证后台。

注

一旦用户被认证过，Django会在用户的session中存储他使用的认证后端，然后在session有效期中一直会为该用户提供此后端认证。这种高效意味着验证源被缓存基于per-session基础, 所以如果你改变 `AUTHENTICATION_BACKENDS` , 如果你需要迫使用户重新认证，需要清除掉 session 数据.一个简单的方式是使用这个方法：`Session.objects.all().delete()` .

编写一个认证后端

一个认证后端是个实现两个方法的类: `get_user(user_id)` and `authenticate(**credentials)` , as well as a set of optional permission related *authorization methods*.

`get_user` 方法要求一个参数 `user_id` –这个参数可以是用户名，数据库中的ID或其它标识 `User` 对象的主键– 方法返回一个 `User` 对象.

`身份验证` 方法使用凭据作为关键字参数。大多数情况下，代码如下：

```
class MyBackend(object):
    def authenticate(self, username=None, password=None):
        # Check the username/password and return a User.
        ...
```

当然，它也可以接收token的方式作为参数，例如:

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a User.
        ...
```

不管怎样, `authenticate` 至少应该检查凭证, 如果凭证合法，它应该返回一个匹配于登录信息的 `User` 实例。如果不合法，则返回 `None` .

正如文档开头所描述的，Django的 admin 与Django `User` 对象是紧耦合的。到目前为止, 最好的解决方法是给每一个在你后台的用户创建一个 `User` 对象 (e.g., in your LDAP directory, your external SQL database, etc.) 你可以先写一个脚本来做

这件事，或者用你的 `authenticate` 方法在用户登陆的时候完成这件事。

这里有一个例子，后台对你定义在 `settings.py` 文件里的用户和密码进行验证，并且在用第一次验证的时候创建一个 `User` 对象:

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend(object):
    """
 Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWOR
D.

 Use the login name, and a hash of the password. For example:

 ADMIN_LOGIN = 'admin'
 ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc3
3051de'
 """

    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASS
WORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. Note that we can set passwo
rd
                # to anything, because it won't be checked; the
password
                # from settings.py will.
                user = User(username=username, password='get fro
m settings.py')
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

在定制后端中处理授权

自定义验证后端能提供自己的权限。

当认证后端完成了这些功能 (`get_group_permissions()` , `get_all_permissions()` , `has_perm()` , and `has_module_perms()`) 那么user model就会给它授予相对应的许可。

提供给用户的权限是所有后端返回的所有权限的超集也就是说，只要任意一个backend授予了一个user权限，django就给这个user这个权限。

New in Django 1.8:

如果后端在 `has_perm()` 或 `has_module_perms()` 中引发 `PermissionDenied` 异常，授权将立即失败，Django不会检查后端接下来。

上述的简单backend可以相当容易的完成授予admin权限。

```
class SettingsBackend(object):
    ...
    def has_perm(self, user_obj, perm, obj=None):
        if user_obj.username == settings.ADMIN_LOGIN:
            return True
        else:
            return False
```

在上例中，授予了用户所有访问权限。注意， 由于 `django.contrib.auth.models.User` 同名函数将接收同样的参数，认证后台接收到的 user_obj，有可能是匿名用户 anonymous

一个完整的认证过程，可以参考 `ModelBackend` 类，它位于django/contrib/auth/backends.py，ModelBackend是默认的认证后台，并且大多数情况下会对 `auth_permission` 表进行查询。 如果你想对后台API提供自定义行为，你可以利用Python继承的优势，继承 `ModelBackend` 并自定义后台API

授权匿名用户

匿名用户是指不经过身份验证即他们有没有提供有效的身份验证细节。然而，这并不一定意味着他们不被授权做任何事情。在最基本的层面上，大多数网站授权匿名用户浏览大部分网站，许多网站允许匿名张贴评论等。

Django 的权限框架没有一个地方来存储匿名用户的权限。然而，传递给身份验证后端的用户对象可能是 `django.contrib.auth.models.AnonymousUser` 对象，该对象允许后端指定匿名用户自定义的授权行为。这对可重用应用的作者是很有用的, 因为他可以委托所有的请求, 例如控制匿名用户访问,给这个认证后端, 而不需要设置它

非活动用户的授权

非活动用户是经过身份验证但属性 `is_active` 设置为 `False` 的用户。然而这并不意味着他们无权做任何事情。例如他们可以被允许激活他们的帐户。

对权限系统中的匿名用户的支持允许匿名用户具有执行某些操作的权限的情况，而未被认证的用户不具有。

不要忘记在自己的后端权限方法中测试用户的 `is_active` 属性。

操作对象权限

django的权限框架对对象权限有基础的支持, 尽管在它的核心没有实现它.这意味着对象权限检查将始终返回 `False` 或空列表 （取决于检查的行为） 。一个认证后端将传递关键字参数 `obj` 和 `user_obj` 给每一个对象相关的认证方法, 并且能够返回适当的对象级别的权限.

自定义权限

要为给定模型对象创建自定义权限，请使用 `权限` 模型元属性。

此示例任务模型创建三个自定义权限，即用户是否可以对您的应用程序任务实例执行操作：

```
class Task(models.Model):
    ...
    class Meta:
        permissions = (
            ("view_task", "Can see available tasks"),
            ("change_task_status", "Can change the status of tas
ks"),
            ("close_task", "Can remove a task by setting its sta
tus as closed"),
        )
```

唯一需要做的就是在运行 `manage.py migrate` 时创建这些额外的权限。当用户尝试访问应用程序提供的功能（查看任务，更改任务状态，关闭任务）时，您的代码负责检查这些权限的值。继续上面的示例，以下检查用户是否可以查看任务：

```
user.has_perm('app.view_task')
```

扩展已有的用户模型

有两种方法来扩展默认的 `User` 模型，而不用替换你自己的模型。 如果你需要的只是行为上的改变，而不需要对数据库中存储的内容做任何改变，你可以创建基于 `User` 的代理模型。代理模型提供的功能包括默认的排序、自定义管理器以及自定义模型方法。

如果你想存储新字段到已有的 `User` 里，那么你可以选择*one-to-one relationship*来扩展用户信息。这种 one-to-one 模型一般被称为资料模型(profile model)，它通常被用来存储一些有关网站用户的非验证性（ non-auth ）资料。例如，你可以创建一个员工模型 (Employee model)：

```
from django.contrib.auth.models import User

class Employee(models.Model):
    user = models.OneToOneField(User)
    department = models.CharField(max_length=100)
```

假设一个员工Fred Smith 既有User 模型又有Employee 模型，你可以使用Django 标准的关联模型访问相关联的信息：

```
>>> u = User.objects.get(username='fsmith')
>>> freds_department = u.employee.department
```

要将个人资料模型的字段添加到管理后台的用户页面中，请在应用程序的 `admin.py` 定义一个 `InlineModelAdmin` （对于本示例，我们将使用 `StackedInline`)并将其添加到 `UserAdmin` 类并向 `User` 类注册的：

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.models import User

from my_user_profile_app.models import Employee

# Define an inline admin descriptor for Employee model
# which acts a bit like a singleton
class EmployeeInline(admin.StackedInline):
    model = Employee
    can_delete = False
    verbose_name_plural = 'employee'

# Define a new User admin
class UserAdmin(UserAdmin):
    inlines = (EmployeeInline, )

# Re-register UserAdmin
admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

这些Profile models在任何方面都不特殊，它们就是和User model多了一个一对一链接的普通Django models。 这种情形下，它们不会在一名用户创建时自动创建, 但是 `django.db.models.signals.post_save` 可以在适当的时候用于创建或更新相关模型。

注意使用相关模型的成果需另外的查询或者联结来获取相关数据，基于你的需求替换用户模型并添加相关字段可能是你更好的选择。但是，在你项目应用程序中，指向默认用户模型的链接可能带来额外的数据库负载。

重写用户模型

Django 内建的 `User` 模型可能不适合某些类型的项目。例如，在某些网站上使用邮件地址而不是用户名作为身份的标识可能更合理。

Django 允许你通过 `AUTH_USER_MODEL` 设置覆盖默认的User 模型， 其值引用一个自定义的模型。

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

上面的值表示Django 应用的名称（必须位于 `INSTALLED_APPS` 中） 和你想使用的User 模型的名称。

注意

改变 `AUTH_USER_MODEL` 对你的数据库结构有很大的影响。它改变了一些会使用到的表格，并且会影响到一些外键和多对多关系的构造。如果你打算设置 `AUTH_USER_MODEL` , 你应该在创建任何迁移或者第一次运行 `manage.py migrate` 前设置它。

在你有表格被创建后更改此设置是不被 `makemigrations` 支持的，并且会导致你需要手动修改数据库结构，从旧用户表中导出数据，可能重新应用一些迁移。

警告

由于Django的可交换模型的动态依赖特性的局限, 你必须确保 `AUTH_USER_MODEL` 引用的模型在所属app中第一个迁移文件中被创建(通常命名为 `0001_initial`); 否则, 你会碰到错误.

此外，在运行迁移时可能会遇到CircularDependencyError，因为Django由于动态依赖性而无法自动断开依赖性循环。如果您看到此错误，您应该通过将依赖于您的用户模型的模型移动到第二个迁移中来打破循环（您可以尝试制作两个具有ForeignKey的正常模型，并查看 `makemigrations`

可重复使用的应用程式和 `AUTH_USER_MODEL`

可重复使用的应用不应实施自定义用户模型。项目可能使用许多应用程序，并且实施自定义用户模型的两个可重用应用程序不能一起使用。如果您需要在应用中存储每个用户的信息，请使用 `ForeignKey` 或 `OneToOneField` 设置 `settings.` AUTH_USER_MODEL，如下所述。

引用User 模型

在 `AUTH_USER_MODEL` 设置改成其它用户模型的项目中，如果你直接引用 `User` （例如，通过一个外键引用它），你的代码将不能工作。

`get_user_model` ()[source]

你应该使用 `django.contrib.auth.get_user_model()` 来引用用户模型，而不要直接引用 `User` 。这个方法将返回当前正在使用的用户模型 —— 指定的自定义用户模型或者 `User` 。

当你定义一个外键或者到用户模型的多对多关系时，你应该使用 `AUTH_USER_MODEL` 设置来指定自定义的模型。例如：

```
from django.conf import settings
from django.db import models

class Article(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL)
```

New in Django 1.7:

连接用户模型发出的信号时， 应该使用 `AUTH_USER_MODEL` 设置指定自定义的模型。例如：

```
from django.conf import settings
from django.db.models.signals import post_save

def post_save_receiver(sender, instance, created, **kwargs):
    pass

post_save.connect(post_save_receiver, sender=settings.AUTH_USER_
MODEL)
```

一般来说，在导入时候执行的代码中，你应该使用 `AUTH_USER_MODEL` 设置引用用户模型。 `get_user_model()` 只在Django 已经导入所有的模型后才工作。

指定自定义的用户模型

模型设计考虑

处理不直接相关的认证在自定义用户模型信息之前，应仔细考虑。

这可能是更好的存储应用程序特定的用户信息在与用户模式的关系的典范。这使得每一个应用，而不用担心与其他应用程序冲突指定自己的用户数据需求.另一方面，查询来检索此相关的信息将涉及的数据库连接，这可能对性能有影响。

Django 期望你自定义的 User model 满足一些最低要求

1. 模型必须有一个唯一的字段可被用于识别目的。可以是一个用户名，电子邮件

地址，或任何其它独特属性。
2. 你的模型必须提供一种方法可以在"short"and"long"form可以定位到用户。最普遍的方法是用用户的名来作为简称,用用户的全名来作为全称。然而，对这两种方式没有特定的要求,如果你想，他们可以返回完全相同的值。

Changed in Django 1.8:

Django的旧版本要求你的模型有一个整数主键也是如此。

创建一个规范的自定义模型最简单的方法是继承 `AbstractBaseUser` `AbstractBaseUser` 提供 `User` 模型的核心实现，包括散列密码和令牌化密码重置。然后，您必须提供一些关键的实施细节：

class `models.``CustomUser`

`USERNAME_FIELD`

描述User模型上用作唯一标识符的字段名称的字符串。这通常是某种用户名，但它也可以是电子邮件地址或任何其他唯一标识符。字段必须必须是唯一的（即在其定义中设置 `unique=True` ）。

在以下示例中，字段 `identifier` 用作标识字段：

```
class MyUser(AbstractBaseUser):
    identifier = models.CharField(max_length=40, unique=True)
    ...
    USERNAME_FIELD = 'identifier'
```

New in Django 1.8.

`USERNAME_FIELD` 现在支持 `ForeignKey` 。由于在 `createsuperuser` 提示期间没有办法传递模型实例，因此希望用户在默认情况下输入 `to_field` 值（ `primary_key` ）的现有实例。

`REQUIRED_FIELDS`

当通过 `createsuperuser` 管理命令创建一个用户时，用于提示的一个字段名称列表.。将会提示给列表里面的每一个字段提供一个值。 它包含的必须是 为 `False` 或者 `blank` 未定义的字段， 也可包含你想要在交互地创建一个新的用户时想要展示的其他字段。 `REQUIRED_FIELDS` 在Django的其他部分没有任何影响，例如在管理员中创建用户。

例如，以下是定义两个必需字段（出生日期和身高）的 `User` 模型的部分定义：

```
class MyUser(AbstractBaseUser):
    ...
    date_of_birth = models.DateField()
    height = models.FloatField()
    ...
    REQUIRED_FIELDS = ['date_of_birth', 'height']
```

注意

`REQUIRED_FIELDS` 必须包含 `User` 模型中的所有必填字段，但应不应包含 `USERNAME_FIELD` 或 `password` ，因为将始终提示输入这些字段。

New in Django 1.8.

`REQUIRED_FIELDS` 现在支持 `ForeignKey` 。由于在 `createsuperuser` 提示期间没有办法传递模型实例，因此希望用户在默认情况下输入 `to_field` 值（ `primary_key` ）的现有实例。

`is_active`

指示用户是否被视为“活动”的布尔属性。此属性作为 `AbstractBaseUser` 上的属性提供，默认为 `True` 。如何选择实施它将取决于您选择的身份验证后端的详细信息。请参阅 `is_active attribute on the built-in user model` 。

`get_full_name` ()

用户更长且正式的标识.常见的解释会是用户的完整名称，但它可以是任何字符串，用于标识用户。

`get_short_name` ()

一个短的且非正式用户的标识符。常见的解释会是第一个用户的名称，但它可以是任意字符串，用于以非正式的方式标识用户。它也可能会返回与 `django.contrib.auth.models.User.get_full_name()` 相同的值。

以下方法适用于 `AbstractBaseUser` 的任何子类：

class `models.``AbstractBaseUser`

`get_username` ()

返回由 `USERNAME_FIELD` 指定的字段的值。

`is_anonymous` ()

始终返回 `False` 。这是区分 `AnonymousUser` 对象的一种方法。通常，您应该优先使用 `is_authenticated()` 到此方法。

`is_authenticated` ()

始终返回 `True` 。这是一种判断用户是否已通过身份验证的方法。这并不意味着任何权限，并且不检查用户是否处于活动状态 - 它仅指示用户已提供有效的用户名和密码。

`set_password` (*raw_password*)

将用户的密码设置为给定的原始字符串，注意密码哈希。不保存 `AbstractBaseUser` 对象。

当raw_password为 `None` 时，密码将被设置为不可用的密码，如同使用 `set_unusable_password()` 。

`check_password` (*raw_password*)

如果给定的原始字符串是用户的正确密码，则返回 `True` 。（这将在进行比较时处理密码散列。）

`set_unusable_password` ()

将用户标记为没有设置密码。这与为密码使用空白字符串不同。 `check_password()` 此用户将永远不会返回 `True` 。不保存 `AbstractBaseUser` 对象。

如果针对现有外部源（例如LDAP目录）进行应用程序的身份验证，则可能需要这样做。

`has_usable_password` ()

如果 `set_unusable_password()` 已为此用户调用，则返回 `False` 。

`get_session_auth_hash` ()

New in Django 1.7.

返回密码字段的HMAC。用于*Session invalidation on password change*。

你应该再为你的 `User` 模型自定义一个管理器。 如果你的 `User` 模型定义了这些字段： `username` , `email` , `is_staff` , `is_active` , `is_superuser` , `last_login` , and `date_joined` 跟默认的 `User` 的字段是一样的话，那么你就使用Django的 `UserManager` 就行了;总之，如果你的 `User` 定义了不同的字段，你就要去自定义一个管理器，它继承自 `BaseUserManager` 并提供两个额外的方法:

class `models.``CustomUserManager`

`create_user` (*username_field*, *password=None*, ***other_fields*)

`create_user()` 原本接受username，以及其它所有必填字段作为参数。例如，如果你的user模型使用 `email` 作为username字段，并且使用 `date_of_birth` 作为必填字段，那么 `create_user` 应该定义为：

```
def create_user(self, email, date_of_birth, password=None):
    # create user here
    ...
```

`create_superuser` (*username_field*, *password*, ***other_fields*)

`create_superuser()` 的原型应该接受用户名字段，以及所有必需的字段作为参数。例如，如果您的用户模型使用 `email` 作为用户名字段，并且 `date_of_birth` 为必填字段，则 `create_superuser` 应定义为：

```
def create_superuser(self, email, date_of_birth, password):
    # create superuser here
    ...
```

与 `create_user()` 不同， `create_superuser()` 必须要求调用方提供密码。

`BaseUserManager` 提供以下实用程序方法：

class `models.``BaseUserManager`

`normalize_email` (*email*)

一个通过将部分电子邮箱地址转为小写来使其规范化的类方法 `classmethod`

`get_by_natural_key` (*username*)

使用由 `USERNAME_FIELD` 指定的字段内容检索用户实例。

`make_random_password` (*length=10, allowed_chars='abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ23456789'*)

返回具有给定长度和给定字符串的允许字符的随机密码。请注意，默认值 `allowed_chars` 不包含可能导致用户混淆的字母，包括：

- `i` ， `l` ， `I` 和 `1` （小写字母i，小写字母L，大写字母i，第一号）
- `o` ， `O` 和 `0` （小写字母o，大写字母o和零）

Extending Django's default User

如果你完全满意Django 的 `用户` 模型且你只想添加一些额外的配置文件信息，您可以简单的继承 `django.contrib.auth.models.AbstractUser` 并添加您的自定义字段，尽管我们建议像"Model design considerations"中描述的 *Specifying a custom User model*那样，使用一个单独的模型。 `AbstractUser` 作为一种 抽象模型 提供默认 `用户` 的完整实现。

自定义用户与内置身份验证表单

正如您所期望的，内置Django的*forms*和*views*对他们正在使用的用户模型做出某些假设。

如果您的用户模型不遵循相同的假设，可能需要定义替换表单，并作为auth视图配置的一部分传递该表单。

- `UserCreationForm`

  取决于 `User` 模型。必须为任何自定义用户模型重写。

- `UserChangeForm`

取决于 `User` 模型。必须为任何自定义用户模型重写。

- `AuthenticationForm`

  与 `AbstractBaseUser` 的任何子类一起使用，并将适应使用 `USERNAME_FIELD` 中定义的字段。

- `PasswordResetForm`

  假设用户模型具有可用于标识用户的 `email` 字段和名为 `is_active` 的布尔字段，以防止对非活动用户进行密码重置。

- `SetPasswordForm`

  适用于 `AbstractBaseUser` 的任何子类

- `PasswordChangeForm`

  适用于 `AbstractBaseUser` 的任何子类

- `AdminPasswordChangeForm`

  适用于 `AbstractBaseUser` 的任何子类

自定义用户和 `django.contrib.admin`

如果你想让你自定义的User模型也可以在站点管理上工作，那么你的模型应该再定义一些额外的属性和方法。 这些方法允许管理员去控制User到管理内容的访问:

class `models.``CustomUser`

`is_staff`

如果允许用户访问管理网站，则返回 `True` 。

`is_active`

如果用户帐户当前处于活动状态，则返回 `True` 。

`has_perm(perm, obj=None):`

如果用户具有命名权限，则返回 `True` 。如果提供 `obj` ，则需要针对特定对象实例检查权限。

`has_module_perms(app_label):`

如果用户有权访问给定应用中的模型，则返回 `True` 。

您还需要向管理员注册您的自定义用户模型。如果您的自定义User模型扩展 `django.contrib.auth.models.AbstractUser` ，则可以使用Django现有的 `django.contrib.auth.admin.UserAdmin` 类。但是，如果您的User模型扩展 `AbstractBaseUser` ，则需要定义一个自定义的 `ModelAdmin` 类。可以将默

认 `django.contrib.auth.admin.UserAdmin` ；但是，您需要覆盖引用 `django.contrib.auth.models.AbstractUser` 上不在您的自定义User类上的字段的任何定义。

自定义用户和权限

为了方便将Django的权限框架包含到你自己的User类中，Django提供了 `PermissionsMixin` 。这是一个抽象模型，您可以包含在用户模型的类层次结构中，为您提供支持Django权限模型所需的所有方法和数据库字段。

`PermissionsMixin` 提供了以下方法和属性：

class `models.``PermissionsMixin`

`is_superuser`

布尔值。指定此用户具有所有权限，而不显式分配它们。

`get_group_permissions` (*obj=None*)

通过用户的组返回用户拥有的一组权限字符串。

如果传入 `obj` ，则仅返回此特定对象的组权限。

`get_all_permissions` (*obj=None*)

通过组和用户权限返回用户拥有的一组权限字符串。

如果传入 `obj` ，则仅返回此特定对象的权限。

`has_perm` (*perm*, *obj=None*)

如果用户具有指定的权限，则返回 `True` ，其中 `perm` 的格式为 `“&lt； app label& &lt； permission codename&gt；“` （请参阅*permissions*）。如果用户处于非活动状态，此方法将始终返回 `False` 。

如果传入 `obj` ，此方法将不会检查模型的权限，而是检查此特定对象。

`has_perms` (*perm_list*, *obj=None*)

如果用户具有每个指定的权限，则返回 `True` ，其中每个perm的格式为 `“&lt； app label&gt；。&lt； permission codename&gt；“` 。如果用户处于非活动状态，此方法将始终返回 `False` 。

如果传入 `obj` ，此方法将不会检查模型的权限，而是检查特定对象。

`has_module_perms` (*package_name*)

如果用户在给定的包（Django应用标签）中有任何权限，则返回 `True` 。如果用户处于非活动状态，此方法将始终返回 `False` 。

ModelBackend

如果您不包含 `PermissionsMixin` ，则必须确保不要调用 `ModelBackend` 上的权限方法。 `ModelBackend` 假定某些字段在您的用户模型上可用。如果您的用户模型未提供这些字段，则在检查权限时将收到数据库错误。

自定义用户和代理模型

自定义用户模型的一个限制是，安装自定义用户模型将破坏扩展 `User` 的任何代理模型。代理模型必须基于具体的基类；通过定义自定义User模型，您可以删除Django可靠地识别基类的能力。

如果项目使用代理模型，则必须修改代理以扩展项目中当前使用的用户模型，或将代理的行为合并到用户子类中。

定制用户和测试/夹具

如果您正在编写与用户模型交互的应用程序，则必须采取一些预防措施，以确保测试套件将运行，而不管项目正在使用的用户模型。如果用户模型已换出，任何实例化User实例的测试都将失败。这包括使用夹具创建User实例的任何尝试。

为确保您的测试套件能够在任何项目配置中传递， `django.contrib.auth.tests.utils` 定义了一个 `@skipIfCustomUser` 装饰器。如果正在使用除默认Django用户以外的任何用户模型，此装饰器将导致跳过测试用例。这个装饰器可以应用于单个测试或整个测试类。

根据您的应用程序，还可能需要添加测试，以确保应用程序与任何用户模型配合使用，而不仅仅是默认的用户模型。为了帮助这个，Django提供了两个可以在测试套件中使用的替代用户模型：

class `tests.custom_user.``CustomUser`

使用 `email` 字段作为用户名的自定义用户模型，并且具有基本的管理员兼容的权限设置

class `tests.custom_user.``ExtensionUser`

扩展 `django.contrib.auth.models.AbstractUser` 的自定义用户模型，添加了 `date_of_birth` 字段。

然后，您可以使用 `@override_settings` 装饰器使该测试与自定义User模型一起运行。例如，这里是一个测试的骨架，它将测试三个可能的用户模型 - 默认值，加上 `auth` app提供的两个用户模型：

```
from django.contrib.auth.tests.utils import skipIfCustomUser
from django.contrib.auth.tests.custom_user import CustomUser, Ex
tensionUser
from django.test import TestCase, override_settings

class ApplicationTestCase(TestCase):
    @skipIfCustomUser
    def test_normal_user(self):
        "Run tests for the normal user model"
        self.assertSomething()

    @override_settings(AUTH_USER_MODEL='auth.CustomUser')
    def test_custom_user(self):
        "Run tests for a custom user model with email-based auth
entication"
        self.assertSomething()

    @override_settings(AUTH_USER_MODEL='auth.ExtensionUser')
    def test_extension_user(self):
        "Run tests for a simple extension of the built-in User."
        self.assertSomething()
```

一个完整例子

这是一个管理器允许的自定义user这个用户模型使用邮箱地址作为用户名，并且要求填写出生年月。它不提供任何权限检查，超出了用户帐户上的一个简单的 `admin` 标志。此模型将与所有内置的身份验证表单和视图兼容，但用户创建表单除外。此示例说明大多数组件如何协同工作，但不打算直接复制到项目以供生产使用。

此代码将全部位于自定义身份验证应用程序的 `models.py` 文件中：

```
from django.db import models
from django.contrib.auth.models import (
    BaseUserManager, AbstractBaseUser
)

class MyUserManager(BaseUserManager):
    def create_user(self, email, date_of_birth, password=None):
        """
 Creates and saves a User with the given email, date of
 birth and password.
 """
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
            date_of_birth=date_of_birth,
```

```
        )

        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, date_of_birth, password):
        """
 Creates and saves a superuser with the given email, date of
 birth and password.
 """
        user = self.create_user(email,
            password=password,
            date_of_birth=date_of_birth
        )
        user.is_admin = True
        user.save(using=self._db)
        return user

class MyUser(AbstractBaseUser):
    email = models.EmailField(
        verbose_name='email address',
        max_length=255,
        unique=True,
    )
    date_of_birth = models.DateField()
    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)

    objects = MyUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['date_of_birth']

    def get_full_name(self):
        # The user is identified by their email address
        return self.email

    def get_short_name(self):
        # The user is identified by their email address
        return self.email

    def __str__(self):              # __unicode__ on Python 2
        return self.email

    def has_perm(self, perm, obj=None):
        "Does the user have a specific permission?"
        # Simplest possible answer: Yes, always
        return True

    def has_module_perms(self, app_label):
        "Does the user have permissions to view the app `app_lab
el`?"
```

```
        # Simplest possible answer: Yes, always
        return True

    @property
    def is_staff(self):
        "Is the user a member of staff?"
        # Simplest possible answer: All admins are staff
        return self.is_admin
```

然后，要使用Django的管理员注册此自定义User模型，应用程序的 `admin.py` 文件中需要以下代码：

```
from django import forms
from django.contrib import admin
from django.contrib.auth.models import Group
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.forms import ReadOnlyPasswordHashField

from customauth.models import MyUser

class UserCreationForm(forms.ModelForm):
    """A form for creating new users. Includes all the required
 fields, plus a repeated password."""
    password1 = forms.CharField(label='Password', widget=forms.P
asswordInput)
    password2 = forms.CharField(label='Password confirmation', w
idget=forms.PasswordInput)

    class Meta:
        model = MyUser
        fields = ('email', 'date_of_birth')

    def clean_password2(self):
        # Check that the two password entries match
        password1 = self.cleaned_data.get("password1")
        password2 = self.cleaned_data.get("password2")
        if password1 and password2 and password1 != password2:
            raise forms.ValidationError("Passwords don't match")
        return password2

    def save(self, commit=True):
        # Save the provided password in hashed format
        user = super(UserCreationForm, self).save(commit=False)
        user.set_password(self.cleaned_data["password1"])
        if commit:
            user.save()
        return user

class UserChangeForm(forms.ModelForm):
    """A form for updating users. Includes all the fields on
 the user, but replaces the password field with admin's
```

```
 password hash display field.
 """
    password = ReadOnlyPasswordHashField()

    class Meta:
        model = MyUser
        fields = ('email', 'password', 'date_of_birth', 'is_acti
ve', 'is_admin')

    def clean_password(self):
        # Regardless of what the user provides, return the initi
al value.
        # This is done here, rather than on the field, because t
he
        # field does not have access to the initial value
        return self.initial["password"]

class MyUserAdmin(UserAdmin):
    # The forms to add and change user instances
    form = UserChangeForm
    add_form = UserCreationForm

    # The fields to be used in displaying the User model.
    # These override the definitions on the base UserAdmin
    # that reference specific fields on auth.User.
    list_display = ('email', 'date_of_birth', 'is_admin')
    list_filter = ('is_admin',)
    fieldsets = (
        (None, {'fields': ('email', 'password')}),
        ('Personal info', {'fields': ('date_of_birth',)}),
        ('Permissions', {'fields': ('is_admin',)}),
    )
    # add_fieldsets is not a standard ModelAdmin attribute. User
Admin
    # overrides get_fieldsets to use this attribute when creatin
g a user.
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('email', 'date_of_birth', 'password1', 'p
assword2')}
        ),
    )
    search_fields = ('email',)
    ordering = ('email',)
    filter_horizontal = ()

# Now register the new UserAdmin...
admin.site.register(MyUser, MyUserAdmin)
# ... and, since we're not using Django's built-in permissions,
# unregister the Group model from admin.
admin.site.unregister(Group)
```

最后，使用 `settings.py` 中的 `AUTH_USER_MODEL` 设置将自定义模型指定为项目的默认用户模型：

```
AUTH_USER_MODEL = 'customauth.MyUser'
```

匿名用户

class `models.``AnonymousUser`

`django.contrib.auth.models.AnonymousUser` 类实现了 `django.contrib.auth.models.User` 接口，但具有下面几个不同点：

- *id* 永远为 `None` 。
- `username` 永远为空字符串。
- `get_username()` 永远返回空字符串。
- `is_staff` 和 `is_superuser` 永远为 `False` 。
- `is_active` 永远为 `False` 。
- `groups` 和 `user_permissions` 永远为空。
- `is_anonymous()` 返回 `True` 而不是 `False` 。
- `is_authenticated()` 返回 `False` 而不是 `True` 。
- `set_password()` 、 `check_password()` 、 `save()` 和 `delete()` 引发 `NotImplementedError` 。

New in Django 1.8:

新增 `AnonymousUser.get_username()` 以更好地模拟 `django.contrib.auth.models.User` 。

在实际应用中，你自己可能不需要使用 `AnonymousUser` 对象，它们用于Web 请求，在下节会讲述。

django.contrib.auth

这份文档提供Django 认证系统组件的API 参考资料。对于这些组件的用法以及如何自定义认证和授权请参照认证主题的相关指南。

用户

字段

class `models.``User`

`User` 对象具有如下字段：

`username`

必选。少于等于30个字符。 用户名可以包含字母、数字、`_` 、`@` 、`+` 、`.` 和 `-` 字符。

`first_name`

可选。 少于等于30个字符。

`last_name`

可选。少于30个字符。

`email`

可选。邮箱地址。

`password`

必选。 密码的哈希及元数据。（Django 不保存原始密码）。原始密码可以无限长而且可以包含任意字符。参见密码相关的文档。

`groups`

与 `Group` 之间的多对多关系。

`user_permissions`

与 `Permission` 之间的多对多关系。

`is_staff`

布尔值。指示用户是否可以访问Admin 站点。

`is_active`

布尔值。指示用户的账号是否激活。我们建议把这个标记设置为 `False` 来代替删除账号；这样的话，如果你的应用和User 之间有外键关联，外键就不会失效。

它不是用来控制用户是否能够登录。认证的后端没有要求检查 `is_active` 标记，而且默认的后端不会检查。如果你想在 `is_active` 为 `False` 时拒绝用户登录，你需要在你自己的视图或自定义的认证后端中作检查。但是，默认的 `login()` 视图使用的 `AuthenticationForm` 却会作这个检查，正如在Django 的Admin 站点中所做的权限检查方法如 `has_perm()` 和认证一样。对于未激活的用户，所有这些函数/方法都返回 `False` 。

`is_superuser`

布尔值。指定这个用户拥有所有的权限而不需要给他们分配明确的权限。

`last_login`

用户最后一次登录的时间。

Changed in Django 1.8:

如果这个用户没有登录过，这个字段将会是 `null` 。以前默认设置成当前的date/time。

`date_joined`

账户创建的时间。当账号创建时，默认设置为当前的date/time。

方法

class `models.``User`

`get_username` ()

返回这个User 的username。因为User 模型可以置换，你应该使用这个方法而不要直接访问username 属性。

`is_anonymous` ()

永远返回 `False` 。这是区别 `User` 和 `AnonymousUser` 对象的一种方法。一般情况下，相比这个方法更建议你使用 `is_authenticated()` 。

`is_authenticated` ()

永远返回 `True` （与 `AnonymousUser.is_authenticated()` 永远返回 `False` 相反）。这是区分用户是否已经认证的一种方法。它不检查权限、用户是否激活以及是否具有一个合法的会话。即使通常你将在 `request.user` 上面 调用这个方法来确认用户是否已经被 `AuthenticationMiddleware` 填充（表示当前登录的用户），你应该明白这个方法对于任何 `User` 实例都返回 `True` 。

`get_full_name` ()

返回 `first_name` 和 `last_name` ，之间带有一个空格。

`get_short_name` ()

返回 `first_name` 。

`set_password` (*raw_password*)

设置用户的密码为给定的原始字符串，并负责密码的哈希。不会保存 `User` 对象。

当 `raw_password` 为 `None` 时，密码将设置为一个不可用的密码，和使用 `set_unusable_password()` 的效果一样。

`check_password` (*raw_password*)

如果给出的原始字符串是用户正确的密码，则返回 `True` 。（它负责在比较时密码的哈希）。

`set_unusable_password` ()

标记用户为没有设置密码。它与密码为空的字符串不一样。 `check_password()` 对这种用户永远不会返回 `True` 。不会保存 `User` 对象。

如果你的认证发生在外部例如LDAP 目录时，可能需要这个函数。

`has_usable_password` ()

如果对这个用户调用过 `set_unusable_password()` ，则返回 `False` 。

`get_group_permissions` (*obj=None*)

返回一个用户当前拥有的权限的set，通过用户组

如果传入 `obj` ，则仅返回此特定对象的组权限。http://python.usyiyi.cn/translate/django_182/ref/contrib/auth.html#

`get_all_permissions` (*obj=None*)

通过组和用户权限返回用户拥有的一组权限字符串。

如果传入 `obj` ，则仅返回此特定对象的权限。

`has_perm` (*perm*, *obj=None*)

如果用户具有指定的权限，则返回 `True` ，其中perm的格式为 `“&lt； app label&gt；。&lt； permission codename&gt；“` 。（请参阅有关*permissions*）。如果用户处于非活动状态，此方法将始终返回 `False` 。

如果传入 `obj` ，此方法将不会检查模型的权限，而是检查此特定对象。

`has_perms` (*perm_list*, *obj=None*)

如果用户具有每个指定的权限，则返回 `True` ，其中每个perm的格式为 `“&lt； app label&gt；。&lt； permission codename&gt；“` 。如果用户处于非活动状态，此方法将始终返回 `False` 。

如果传入 `obj` ，此方法将不会检查模型的权限，而是检查特定对象。

`has_module_perms` (*package_name*)

如果用户具有给出的package（Django 的应用标签）中的权限，则返回 `True` 。如果用户没有激活，这个方法将永远返回 `False` 。

`email_user` (*subject*, *message*, *from_email=None*, ***kwargs*)

发生邮件给这个用户。如果 `from_email` 为 `None` ，Django 将使用 `DEFAULT_FROM_EMAIL` 。

Changed in Django 1.7:

任何 `**kwargs` 都将传递给底层的 `send_mail()` 调用。

管理器方法

class ``` models.``UserManager ```

`User` 模型有一个自定义的管理器，它具有以下辅助方法（除了 `BaseUserManager` 提供的方法之外）：

`create_user` (*username*, *email=None*, *password=None*, ***extra_fields*)

创建、保存并返回一个 `User` 。

`username` 和 `password` 设置为给出的值。 `email` 的域名部分将自动转换成小写，返回的 `User` 对象将设置 `is_active` 为 `True` 。

如果没有提供password，将调用 `set_unusable_password()` 。

`extra_fields` 关键字参数将传递给 `User` 的 `__init__` 方法，以允许设置自定义*User*模型 的字段。

参见创建用户 中的示例用法。

`create_superuser` (*username*, *email*, *password*, ***extra_fields*)

与 `create_user()` 相同，但是设置 `is_staff` 和 `is_superuser` 为 `True` 。

匿名用户

class ``` models.``AnonymousUser ```

`django.contrib.auth.models.AnonymousUser` 类实现了 `django.contrib.auth.models.User` 接口，但具有下面几个不同点：

- *id* 永远为 `None` 。
- `username` 永远为空字符串。
- `get_username()` 永远返回空字符串。
- `is_staff` 和 `is_superuser` 永远为 `False` 。
- `is_active` 永远为 `False` 。
- `groups` 和 `user_permissions` 永远为空。
- `is_anonymous()` 返回 `True` 而不是 `False` 。
- `is_authenticated()` 返回 `False` 而不是 `True` 。
- `set_password()` 、 `check_password()` 、 `save()` 和 `delete()` 引发 `NotImplementedError` 。

New in Django 1.8:

新增 `AnonymousUser.get_username()` 以更好地模拟 `django.contrib.auth.models.User` 。

在实际应用中，你自己可能不需要使用 `AnonymousUser` 对象，它们用于Web 请求，在下节会讲述。

权限

class `models.``Permission`

字段

`Permission` 对象有以下字段:

class `models.``Permission`

`name`

必填项. 255个字符或者更少. 例如: `'Can vote'` .

Changed in Django 1.8:

`max_length` 属性从50个字符增加至255个字符

`content_type`

必填项.对 `django_content_type` 数据库表的引用，其中包含每个已安装模型的记录。

`codename`

必须项.小于等于是100个字符.例如: `'can_vote'` .

方法

`Permission` 对象具有类似任何其他*Django model*的标准数据访问方法。

用户群组 Group

class `models.``Group`

字段

`Group` 对象有以下字段:

class `models.``Group`

`name`

必填项，80个字符以内。允许任何字符. 例如: `'Awesome Users'`.

`permissions`

多对多字段到 `Permission` :

```
group.permissions = [permission_list]
group.permissions.add(permission, permission, ...)
group.permissions.remove(permission, permission, ...)
group.permissions.clear()
```

登陆和注销标识

auth框架使用以下*signals*，可用于在用户登录或注销时通知。

`user_logged_in` ()

当用户成功登录时发送。

与此信号一起发送的参数：

`sender`

The class of the user that just logged in.

`request`

The current `HttpRequest` instance.

`user`

The user instance that just logged in.

`user_logged_out` ()

在调用logout方法时发送。

`sender`

As above: the class of the user that just logged out or `None` if the user was not authenticated.

`request`

The current `HttpRequest` instance.

`user`

The user instance that just logged out or `None` if the user was not authenticated.

`user_login_failed` ()

当用户登录失败时发送

`sender`

The name of the module used for authentication.

`credentials`

A dictionary of keyword arguments containing the user credentials that were passed to `authenticate()` or your own custom authentication backend. Credentials matching a set of 'sensitive' patterns, (including password) will not be sent in the clear as part of the signal.

认证使用的后台

这一节详细讲述Django 自带的认证后台。关于如何使用它们以及如何编写你自己的认证后台，参见用户认证指南 中的其它认证源一节。

可用的认证后台

以下是 `django.contrib.auth.backends` 中可以使用的后台：

class `ModelBackend` [source]

这是Django使用的默认认证后台。它使用由用户标识和密码组成的凭据进行认证。对于Django的默认用户模型，用户的标识是用户名，对于自定义的用户模型，它通过USERNAME*FIELD* 字段表示（参见[自定义Users 和认证_] (../../topics/auth/customizing.html)）。

它还处理 `User` 和 `PermissionsMixin` 定义的权限模型。

`has_perm()`，`get_all_permissions()`，`get_user_permissions()`，和 `get_group_permissions()` 允许一个对象作为特定权限参数来传递, 如果条件是 if `obj is not None`. 后端除了返回一个空的permissions 外，并不会去完成他们。

`authenticate` (*username=None*, *password=None*, ***kwargs*)[source]

通过调用 `User.check_password` 验证 `username` 和 `password` 。如果 `username` 没有提供，它会使用 `CustomUser.USERNAME_FIELD` 关键字从 `kwargs` 中获取username。返回一个认证过的User 或 `None` 。

`get_user_permissions` (*user_obj*, *obj=None*)[source]

New in Django 1.8.

返回 `user_obj` 具有的自己用户权限的权限字符串集合。如果 `is_anonymous()` 或 `is_active` 为 `False` ，则返回空集。

`get_group_permissions` (*user_obj*, *obj=None*)[source]

返回 `user_obj` 从其所属组的权限中获取的权限字符集。如果 `is_anonymous()` 或 `is_active` 为 `False` ，则返回空集。

`get_all_permissions` (*user_obj*, *obj=None*)[source]

返回 `user_obj` 的权限字符串集，包括用户权限和组权限。如果 `is_anonymous()` 或 `is_active` 为 `False` ，则返回空集。

`has_perm` (*user_obj*, *perm*, *obj=None*)[source]

使用 `get_all_permissions()` 检查 `user_obj` 是否具有权限字符串 `perm` 。如果用户不是 `is_active` ，则返回 `False` 。

`has_module_perms` (*self*, *user_obj*, *app_label*)[source]

返回 `user_obj` 是否对应用 `app_label` 有任何权限。

class `RemoteUserBackend` [source]

使用这个后台来处理Django的外部认证。. 它使用 `request.` 里面的usernames来进行验证。META ['REMOTE*USER']*。请参阅*[_Authenticating against REMOTE_USER*](../../howto/auth-remote-user.html)文档。

如果你需要更多的控制，你可以创建你自己的验证后端，继承这个类，并重写这些属性或方法：

`RemoteUserBackend.``create_unknown_user`

`True` 或 `False` 。决定是否有一个 `User` 对象已经在数据库中创建。默认为 `True` 。

`RemoteUserBackend.``authenticate` (*remote_user*)[source]

作为 `remote_user` 传递的用户名被认为是可信的。此方法仅返回给定用户名的 `User` 对象，如果 `create_unknown_user` 为 `True` ，则创建新的 `User` 对象。

如果 `create_unknown_user` 是 `False` ，并且在数据库中找不到具有给定用户名的 `User` 对象，则返回 `None` 。

`RemoteUserBackend.``clean_username` (*username*)[source]

Performs any cleaning on the `username` (e.g. stripping LDAP DN information) prior to using it to get or create a `User` object. 返回已清除的用户名。

`RemoteUserBackend.``configure_user` (*user*)[source]

配置新创建的用户。此方法在创建新用户后立即调用，并可用于执行自定义设置操作，例如根据LDAP目录中的属性设置用户的组。返回用户对象。

Django’s cache framework

一个动态网站的基本权衡点就是，它是动态的。 每次用户请求一个页面，Web服务器将进行所有涵盖数据库查询到模版渲染到业务逻辑的请求，用来创建浏览者需要的页面。从开销处理的角度来看，这比你读取一个现成的标准文件的代价要昂贵的多。

对于大多数网络应用程序，这个开销不是很大的问题。我们的应用不是 `washingtonpost.com` or `slashdot.org` ; 他们只是中小型网站，而且只有那么些流量而已。 但对于中等流量的网站来说，尽可能地减少开销是必要的。

这就是需要缓存的地方

缓存一些东西是为了保存那些需要很多计算资源的结果，这样的话就不必在下次重复消耗计算资源。 下面是一些伪代码，用来解释缓存怎样在动态生成的网页中工作的：

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django自带了一个健壮的缓存系统来让你保存动态页面这样避免对于每次请求都重新计算。方便起见，Django提供了不同级别的缓存粒度：你可以缓存特定视图的输出、你可以仅仅缓存那些很难生产出来的部分、或者你可以缓存你的整个网站。

Django也能很好的配合那些“下游”缓存, 比如 Squid 和基于浏览器的缓存。这里有一些缓存你不必要直接去控制但是你可以提供线索， (via HTTP headers)关于你的网站哪些部分需要缓存和如何缓存。

也可以看看

*Cache Framework design philosophy*解释了框架的一些设计决策。

设置缓存

缓存系统需要一些设置才能使用。 也就是说，你必须告诉他你要把数据缓存在哪里- 是数据库中，文件系统或者直接在内存中。 这个决定很重要，因为它会影响你的缓存性能，是的，一些缓存类型要比其他的缓存类型更快速。

你的缓存配置是通过setting 文件的 `CACHES` 配置来实现的。 这里有 `CACHES` 所有可配置的变量值。

Memcached

Django支持的最快，最高效的缓存类型, Memcached 是一个全部基于内存的缓存服务，起初是为了解决LiveJournal.com负载来开发的，后来是由Danga开源出来的。 它被类似Facebook 和 维基百科这种网站使用，用来减少数据库访问，显著的提高了网站的性能。

Memcached 是个守护进程，它被分配了单独的内存块。 它做的所有工作就是为缓存提供一个快速的添加，检索，删除的接口。 所有的数据直接存储在内存中，所以它不能取代数据库或者文件系统的使用。

在安装 Memcached 后， 还需要安装 Memcached 依赖模块。Python 有不少Memcache模块最为常用的是python-memcached and pylibmc两个模块.

需要在Django中使用Memcached时:

- 将 `BACKEND` 设置为 `django.core.cache.backends.memcached.MemcachedCache` 或者 `django.core.cache.backends.memcached.PyLibMCCache` (取决于你所选绑定memcached的方式)
- 将 `LOCATION` 设置为 `ip:port` 值， `ip` 是 Memcached 守护进程的ip地址， `port` 是Memcached 运行的端口。或者设置为 `unix:path` 值， `path` 是 Memcached Unix socket file的路径.

在这个例子中，Memcached 运行再 本地 (127.0.0.1) 的11211端口，使用 `python-memcached` （也就是需要这么一个python插件） 绑定：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

这个例子中，Memcached 通过一个本地的Unix socket file `/tmp/memcached.sock` 来交互，也要使用 `python-memcached` 绑定：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}
```

Memcached有一个非常好的特点就是可以让几个服务的缓存共享。 这就意味着你可以再几个物理机上运行Memcached服务，这些程序将会把这几个机器当做 同一个 缓存，从而不需要复制每个缓存的值在每个机器上。为了使用这个特性，把所有的服务地址放在 `LOCATION` 里面，用分号隔开或者当做一个list。

这个例子，缓存共享在2个Memcached 实例中，IP地址为172.19.26.240 和 172.19.26.242，端口同为11211：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

下面的这个例子，缓存通过下面几个 Memcached 实例共享，IP地址为 172.19.26.240 (端口 11211), 172.19.26.242 (端口 11212), and 172.19.26.244 (端口 11213):

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11212',
            '172.19.26.244:11213',
        ]
    }
}
```

关于Memcached最后要说一点，基于内存的缓存有一个缺点：因为缓存数据是存储在内存中的，所以如果你的服务器宕机数据就会丢失。还要明确， 内存不能替代常驻的数据存储，所以不要把基于内存的缓存当成你唯一的数据存储方式。毫无疑问的，_没有任何的_Django缓存后台应该被用来替代常驻存储--它们要做的是缓存解决方案，而不是存储方案--但是我们在这里指出这一点是因为基于内存的缓存真的是非常的临时。

数据库缓存

Django 可以把缓存保存在你的数据库里。如果你有一个快速的、专业的数据库服务器的话那这种方式是效果最好的。

为了把数据表用来当做你的缓存后台：

- 把 `BACKEND` 设置为 `django.core.cache.backends.db.DatabaseCache`
- 把 `LOCATION` 设置为 `tablename` ，数据表的名称。这个名字可以是任何你想要的名字，只要它是一个合法的表名并且在你的数据库中没有被使用过。

在这个示例中，缓存表的名字是 `my_cache_table` :

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache'
,
        'LOCATION': 'my_cache_table',
    }
}
```

创建缓存表

使用数据库缓存之前，你必须用这个命令来创建缓存表：

```
python manage.py createcachetable
```

这将在你的数据库中创建一个Django的基于数据库缓存系统预期的特定格式的数据表。表名会从 `LOCATION` 中获得。

如果你使用多数据库缓存， `createcachetable` 会在每个缓存中创建一个表。

如果你使用多数据库， `createcachetable` 会遵循你的数据库路由中的 `allow_migrate()` 方法(见下)。

像 `migrate` , `createcachetable` 这样的命令不会碰触现有的表。它只创建非现有的表。

Changed in Django 1.7:

在Django 1.7之前， `createcachetable` 一次只创建一个表。您必须传递要创建的表的名称，如果您使用多个数据库，则必须使用 `--database` 选项。为了向后兼容，这仍然是可能的。

多个数据库

如果你在多数据库的情况下使用数据库缓存，你还必须为你的数据库缓存表设置路由说明。出于路由的目的，数据库缓存表会在一个名为 `django_cache` 的应用下的 `CacheEntry` model中出现。 这个模型不会出现在模型缓存中，但这个模型的细节可以在路由中使用。

例如, 下面的路由会分配所有缓存读操作到 `cache_replica` ，和所有写操作到 `cache_primary` . 缓存表只会同步到 `cache_primary` :

```
class CacheRouter(object):
    """A router to control all database cache operations"""

    def db_for_read(self, model, **hints):
        "All cache read operations go to the replica"
        if model._meta.app_label == 'django_cache':
            return 'cache_replica'
        return None

    def db_for_write(self, model, **hints):
        "All cache write operations go to primary"
        if model._meta.app_label == 'django_cache':
            return 'cache_primary'
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hi
nts):
        "Only install the cache model on primary"
        if app_label == 'django_cache':
            return db == 'cache_primary'
        return None
```

如果你没有指定路由路径给数据库缓存model,那么缓存就会使用 `默认的` 数据库。

当然, 如果你不使用数据库做缓存，你就不需要担心提供路由结构给数据库缓存模型。

文件系统缓存

基于文件的缓存后端序列化和存储每个缓存值作为一个单独的文件。 为了使用这个文件缓存，你要设置 `BACKEND` 为 `"django.core.cache.backends.filebased.FileBasedCache"` 并且 `LOCATION` 设置为一个合适的目录。例如，把缓存储存在 `/var/tmp/django_cache` ,就用这个设置:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBas
edCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

如果你在Windows上，将驱动器号放在路径的开头，如下所示：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBas
edCache',
        'LOCATION': 'c:/foo/bar',
    }
}
```

路径应该是绝对路径– 也就是说，要从你的系统路径开始算。你在末尾添加不添加斜杠都是无所谓的。

请确保，你的路径指向是存在的并且，这个路径下 你有系统用户的足够的读，写权限。继续上面的例子，如果你是一个 名叫 `apache` 用户,确保 `/var/tmp/django_cache` 这个路径存在并且 `apache` 有读和写的权力。

本地内存缓存

这是默认的缓存，如果你不在指定其他的缓存设置。如果你想要具有高速这个有点的基于内存的缓存但是又没有能力带动 Memcached, 那就考虑一下本地缓存吧。这个缓存是per-process（见下文）和线程安全的。要使用它，请将 `BACKEND` 设置为 `"django.core.cache.backends.locmem.LocMemCache"` 。例如：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCach
e',
        'LOCATION': 'unique-snowflake',
    }
}
```

高速缓存 `LOCATION` 用于标识各个存储器存储。如果您只有一个 `locmem` 快取，可以省略 `LOCATION` ；然而，如果您有多个本地内存缓存，您将需要为其中至少一个分配一个名称，以保持它们分离。

注意每个进程都有自己的私有缓存实例，这意味着不可能有跨进程缓存。这显然也意味着本地内存缓存不是特别高效的内存，因此它可能不是生产环境的好选择。这是很好的发展。

虚拟缓存（用于开发）

最后, Django有一个 “dummy” 缓存，而且还不是真正的缓存– 它只是提供了一个缓存接口，但是什么也不做。

如果你有一个生产站点使用重型高速缓存在不同的地方，但一个开发/测试环境中，你不想缓存，不希望有你的代码更改为后面的特殊情况，这非常有用。 要激活虚拟缓存，请设置 `BACKEND` ，如下所示：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache'
,
    }
}
```

使用自定义缓存后端

Django包含一定数量的外部缓存后端的支持，有时你可能想要使用一个自定义的缓存后端。 想要使用外部的缓存，就像python import那样在 `CACHES` 的 `BACKEND` 设置中,就像这样

```
CACHES = {
    'default': {
        'BACKEND': 'path.to.backend',
    }
}
```

如果你建立自己的缓存后端，你可以使用标准的缓存后端作为参考实现。 您可以在Django源的 `django/core/cache/backends/` 目录中找到代码。

注意：没有真正令人信服的原因，例如不支持它们的主机，您应该坚持Django附带的缓存后端。它们经过了良好的测试，易于使用。

Cache 参数

上述每一个缓存后台都可以给定一些额外的参数来控制缓存行为，这些参数在可以在 `CACHES` setting中以额外键值对的形式给定，可以设置的参数如下：

- `TIMEOUT` :缓存的默认过期时间，以秒为单位， 这个参数默认是 `300` seconds (5 分钟).

  New in Django 1.7.

  你可以设置 `TIMEOUT` 为 `None` 这样的话，缓存默认永远不会过期。值设置成 `0` 造成缓存立即失效(缓存就没有意义了)。

- `OPTIONS` : 这个参数应该被传到缓存后端。有效的可选项列表根据缓存的后端不同而不同，由第三方库所支持的缓存将会把这些选项直接配置到底层的缓存库。

  缓存的后端实现自己的选择策略 (i.e., the `locmem` , `filesystem` and `database` backends) 将会履行下面这些选项：

  - `MAX_ENTRIES` :高速缓存允许的最大条目数，超出这个数则旧值将被删除. 这个参数默认是 `300` .

  - `CULL_FREQUENCY`：当达到 `MAX_ENTRIES` 的时候,被删除的条目比率。实际比率是 `1 / CULL_FREQUENCY`，所以设置 `CULL_FREQUENCY` 为 `2` 会在达到 `MAX_ENTRIES` 所设置值时删去一半的缓存。这个参数应该是整数，默认为 `3`.

    把 `CULL_FREQUENCY` 的值设置为 `0` 意味着当达到 `MAX_ENTRIES` 时,缓存将被清空。某些缓存后端 (`database` 尤其)这将以很多缓存丢失为代价,大大*much* 提高接受访问的速度。

- `KEY_PREFIX`：将自动包含（默认情况下预置为）Django服务器使用的所有缓存键的字符串。

  有关详细信息，请参阅*cache documentation*。

- `VERSION`：由Django服务器生成的缓存键的默认版本号。

  有关详细信息，请参阅*cache documentation*。

- `KEY_FUNCTION` 包含函数的虚线路径的字符串，定义如何将前缀，版本和键组成最终缓存键。

  有关详细信息，请参阅*cache documentation*。

在下面这个例子中，一个文件系统缓存后端，缓存过期时间被设置为60秒，最大条目为1000.

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBas
edCache',
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

非法的参数会被忽略掉

站点级缓存

一旦高速缓存设置，最简单的方法是使用缓存缓存整个网站。 你需要把 `'django.middleware.cache.UpdateCacheMiddleware'` 和 `'django.middleware.cache.FetchFromCacheMiddleware'` 添加到你的 `MIDDLEWARE_CLASSES` 设置里, 如例子所示:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

注意

不，这里并没有排版错误：'update'中间件，必须放在列表的开始位置，而fectch中间件，必须放在最后。 细节有点费解，如果您想了解完整内幕请参看下面MIDDLEWARE_CLASSES顺序

然后，添加下面这些需要的参数到settings文件里:

- `CACHE_MIDDLEWARE_ALIAS` – 用于存储的缓存的别名
- `CACHE_MIDDLEWARE_SECONDS` –每个page需要被缓存多少秒.
- `CACHE_MIDDLEWARE_KEY_PREFIX` – 如果缓存被使用相同Django安装的多个网站所共享，那么把这个值设成当前网站名，或其他能代表这个Django实例的唯一字符串，以避免key发生冲突。 如果你不在意的话可以设成空字符串。

`FetchFromCacheMiddleware` 缓存GET和HEAD状态为200的回应，用不同的参数请求相同的url被视为独立的页面，缓存是分开的。这个中间件期望HEAD请求用与相应的GET请求相同的响应头来应答；在这种情况下，它可以返回HEAD请求的缓存的GET响应。

另外， `UpdateCacheMiddleware` 在每个 `HttpResponse` 里自动设置了一些头部信息

- 设置 `Last-Modified` 当一个新(没缓存的)版本的页面被请求时，为当前日期/时间
- 设置 `Expires` 头 为当前日期/时间加上定义的 `CACHE_MIDDLEWARE_SECONDS` .
- 设置 `Cache-Control` 头部来给页面一个最长的有效期, 来自 `CACHE_MIDDLEWARE_SECONDS` 的设置.

查看 *Middleware* 更多中间件.

如果视图设置其自己的缓存到期时间（即，在其 `Cache-Control` 头部中具有 `max-age` 部分），则页面将被缓存直到到期时间，而不是 `CACHE_MIDDLEWARE_SECONDS` 。使用 `django.views.decorators.cache` 中的装饰器可以轻松设置视图的到期时间（使用 `cache_control` 装饰器）或禁用视图的缓存（使用 `never_cache` 装饰器）。有关这些装饰器的更多信息，请参阅使用其他标头部分。

If `USE_I18N` is set to `True` then the generated cache key will include the name of the active *language* – see also *How Django discovers language preference*). 这允许您轻松缓存多语言网站，而无需自己创建缓存密钥。

Cache keys also include the active *language* when `USE_L10N` is set to `True` and the *current time zone* when `USE_TZ` is set to `True`.

单个view缓存

`django.views.decorators.cache.``cache_page` ()

更加轻巧的缓存框架使用方法是对单个有效视图的输出进行缓存。`django.views.decorators.cache` 定义了一个自动缓存视图响应的 `cache_page` 装饰器，使用非常简单:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

`cache_page` 接受一个参数：timeout，秒为单位。在前例中，“ `my_view()` ”视图的结果将被缓存 15 分钟 (注意为了提高可读性我们写了 `60 * 15` . `60 * 15` 等于 `900` —也就是说15分钟等于60秒乘15.)

和站点缓存一样，视图缓存与 URL 无关。如果多个 URL 指向同一视图，每个URL将会分别缓存。 继续 `my_view` 范例，如果 URLconf 如下所示：

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

那么正如你所期待的那样，发送到 `/foo/1/` and `/foo/23/` 会被分别缓存。但是一旦一个明确的 URL (e.g., `/foo/23/`) 已经被请求过了, 之后再度发出的指向该 URL 的请求将使用缓存。

`cache_page` 也可以使用一些额外的参数, `cache` , 指示修饰符去具体使用缓存 (from your `CACHES` setting) 当要缓存页面结果时。默认地, `default` cache 会被使用, 但是你可以特别指定你要用的缓存

```
@cache_page(60 * 15, cache="special_cache")
def my_view(request):
    ...
```

您还可以在每个视图的基础上覆盖缓存前缀。 `cache_page` 采用可选的关键字参数 `key_prefix` ，其工作方式与中间件的 `CACHE_MIDDLEWARE_KEY_PREFIX` 设置相同。它可以这样使用：

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request):
    ...
```

`key_prefix` and `cache` 参数可以被一起指定。将连接 `key_prefix` 参数和 `CACHES` 下指定的 `KEY_PREFIX` 。

在URLconf中指定每个视图的缓存

前一节中的范例将视图硬编码为使用缓存，因为 `cache_page` 在适当的位置对 `my_view` 函数进行了转换。 该方法将视图与缓存系统进行了耦合，从几个方面来说并不理想。例如，你可能想在某个无缓存的站点中重用该视图函数，或者不想通过缓存使用页面的人请求你的页面。 解决这些问题的方法是在 URLconf 中指定视图缓存，而不是在这些视图函数上来指定。

这样做很容易：在URLconf中引用它时，只需用 `cache_page` 包装视图函数。这是以前的URLconf：

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

这也是一样的， `my_view` 包裹在 `cache_page` ：

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```

模板片段缓存

如果想对缓存进行更多的控制，可以使用 `cache` 模板标签来缓存模板的一个片段。 要让模板处理这个标签，把 `{% load cache %}` 放在缓存片段的上面。

标签 `{% cache %}` 将按给定的时间缓存包含块中的内容。它最少需要两个参数：缓存时间（以秒为单位）；给缓存片段起的名称。 该名称将被视为是，不使用变量。例如：

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

有时，你可以依据这个片段内的动态内容缓存多个版本。如上个例子中，可以给站点的每个用户生成不同版本的sidebar缓存。只需要给 `{% cache %}` 标签再传递一个参数来标识区分这个缓存片段。

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

指定一个以上的参数来识别片段是非常好的。 简单的尽可能的传递你需要的参数到 `{% cache %}` 。

如果 `USE_I18N` 设置为 `True` ，则每个网站中间件缓存将*respect the active language*。对于 `cache` 模板标记，您可以使用模板中提供的*translation-specific variables*之一来实现相同的结果：

```
{% load i18n %}
{% load cache %}

{% get_current_language as LANGUAGE_CODE %}

{% cache 600 welcome LANGUAGE_CODE %}
    {% trans "Welcome to example.com" %}
{% endcache %}
```

缓存超时可以是模板变量，只要模板变量解析为整数值即可。例如，如果模板变量 `my_timeout` 设置为值 `600` ，则以下两个示例是等效的：

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

此功能有助于避免模板中的重复。您可以在一个位置设置变量的超时，只需重复使用该值。

New in Django 1.7:

默认情况下，缓存标签将尝试使用名为“template_fragments”的缓存。如果不存在这样的缓存，则它将回退到使用默认缓存。您可以选择要与 `using` 关键字参数一起使用的备用高速缓存后端，该参数必须是标记的最后一个参数。

```
{% cache 300 local-thing ...  using="localcache" %}
```

指定未配置的缓存名称被视为错误。

`django.core.cache.utils.``make_template_fragment_key` (*fragment_name*, *vary_on=None*)

如果您想获得用于缓存片段缓存键，就可以使用 `make_template_fragment_key` . `fragment_name` 就跟 `cache` template tag的第二参数是一样的; `vary_on` 是一个需要传递额外参数的列表。该功能可以用于无效或覆盖缓存项，例如：

```
>>> from django.core.cache import cache
>>> from django.core.cache.utils import make_template_fragment_k
ey
# cache key for {% cache 500 sidebar username %}
>>> key = make_template_fragment_key('sidebar', [username])
>>> cache.delete(key) # invalidates cached template fragment
```

底层的缓存API

有时，缓存整个页面不会让你获益很多，事实上，过度的矫正原来的不方便，变得更加不方便。

也许，例如，您的网站包含的结果取决于几个昂贵的查询，其结果在不同的时间间隔都有所不同。 在这种情况下，使用每个站点或每个视图缓存策略提供的全页缓存是不理想的，因为你不想缓存整个结果（因为一些数据经常变化），但你仍然想要缓存很少变化的结果。

对于这个情况 Django提供了一个底层的 cache API. 你可以用这个 API来储存在缓存中的对象，并且控制粒度随你喜欢。您可以缓存可以安全pickle的任何Python对象：模型对象的字符串，字典，列表等等。 （最常见的Python对象可以Pickle;参考Python文档有关pickle更多信息。）

访问缓存

``` django.core.cache.``caches ```

New in Django 1.7.

你可以通过 类字典对象 `django.core.cache.caches` .访问配置在 `CACHES` 设置中的字典类对象。对同一线程相同的别名重复请求将返回相同的对象。

```
>>> from django.core.cache import caches
>>> cache1 = caches['myalias']
>>> cache2 = caches['myalias']
>>> cache1 is cache2
True
```

如果key不存在，就会引发一个 `InvalidCacheBackendError` 。

为了提供线程安全， 不同的缓存实例将会返回给每一个线程。

``` django.core.cache.``cache ```

作为一个快捷方式，默认缓存可用为 `django.core.cache.cache` ：

```
&gt;&gt;&gt; from django.core.cache import cache
```

此对象等效于 `caches['default']` 。

`django.core.cache.``get_cache` (*backend*, ***kwargs*)

自1.7版起已弃用：此功能已弃用，支持 `caches` 。

在Django 1.7之前，这个函数是获取缓存实例的规范方式。它也可以用于创建具有不同配置的新缓存实例。

```
&gt;&gt;&gt; from django.core.cache import get_cache
&gt;&gt;&gt; get_cache('default')
&gt;&gt;&gt; get_cache('django.core.cache.backends.memcached.MemcachedCache', LOCATION='127.0.0.2')
&gt;&gt;&gt; get_cache('default', TIMEOUT=300)
```

基本用法

The basic interface is `set(key, value, timeout)` and `get(key)` :

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

`timeout` 参数是可选的，默认为 `CACHES` 设置中适当后端的 `timeout` 参数（如上所述）。它是值应该存储在缓存中的秒数。Passing in `None` for `timeout` will cache the value forever. `0` 的 `timeout` 将不会缓存该值。

如果对象不存在于缓存中，则 `cache.get()` 返回 `None` ：

```
# Wait 30 seconds for 'my_key' to expire...

>>> cache.get('my_key')
None
```

我们建议不要在缓存中存储文本值 `None` ，因为您将无法区分您存储的 `None` 值和由返回值表示的缓存未命中 `None` 。

`cache.get()` 可以采用 `default` 参数。如果对象不存在于缓存中，则指定返回哪个值：

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

要添加键（如果它尚不存在），请使用 `add()` 方法。它使用与 `set()` 相同的参数，但如果指定的键已经存在，它不会尝试更新缓存：

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

如果你需要知道 `add()` 是否在缓存中存储了一个值，你可以检查返回值。如果值存储，则返回 `True` ，否则返回 `False` 。

还有一个 `get_many()` 接口，只会命中一次缓存。 `get_many()` 返回一个字典，其中包含您请求的所有实际存在于缓存中的键（并且未过期）：

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

要更有效地设置多个值，请使用 `set_many()` 传递键值对的字典：

```
>>> cache.set_many({'a': 1, 'b': 2, 'c': 3})
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

像 `cache.set()` ， `set_many()` 采用可选的 `timeout` 参数。

您可以使用 `delete()` 显式删除键。这是清除特定对象的缓存的简单方法：

```
>>> cache.delete('a')
```

如果您想一次清除一堆键， `delete_many()` 可以取得要清除的键列表：

```
>>> cache.delete_many(['a', 'b', 'c'])
```

最后，如果要删除缓存中的所有键，请使用 `cache.clear()` 。小心这个； `clear()` 将从缓存中删除所有，而不仅仅是应用程序设置的键。

```
>>> cache.clear()
```

您还可以使用 `incr()` 或 `decr()` 方法分别递增或递减已存在的键。默认情况下，现有高速缓存值将递增或递减1。可以通过向增量/减量调用提供参数来指定其他增量/减量值。如果您尝试增加或减少不存在的缓存键，则会引发ValueError错误：

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

注意

`incr()` / `decr()` 方法不能保证是原子的。在那些支持原子递增/递减（最明显的是memcached后端）的后端，递增和递减操作将是原子的。然而，如果后端本身不提供增量/减量操作，则它将使用两步检索/更新来实现。

如果由缓存后端实现，您可以使用 `close()` 关闭与缓存的连接。

```
>>> cache.close()
```

注意

对于不实现 `close` 方法的缓存，它是一个无操作。

高速缓存密钥前缀

如果您在服务器之间共享缓存实例，或者在您的生产和开发环境之间共享缓存的话，可以使用另一台服务器使用一台服务器缓存的数据。 如果缓存数据的格式是不同的服务器，这可能会导致一些很难诊断问题。

为了防止这种情况，Django提供了由服务器使用的所有前缀缓存键的能力 当保存或检索特定的缓存键时，Django将自动为缓存键加上 `KEY_PREFIX` 缓存设置的值。

通过确保每个Django实例具有不同的 `KEY_PREFIX` ，您可以确保缓存值中没有冲突。

缓存版本控制

当您更改使用缓存值的运行代码时，您可能需要清除任何现有的缓存值。最简单的方法是刷新整个缓存，但是这可能导致缓存值丢失，仍然有效和有用。

Django提供了一种更好的方法来定位各个缓存值。Django的缓存框架具有系统级版本标识符，使用 `VERSION` 缓存设置指定。此设置的值将自动与高速缓存前缀和用户提供的高速缓存密钥相结合，以获取最终的高速缓存密钥。

默认情况下，任何密钥请求都将自动包含站点默认缓存密钥版本。但是，原语缓存函数都包含 `version` 参数，因此可以指定要设置或获取的特定缓存键版本。例如：

```
# Set version 2 of a cache key
>>> cache.set('my_key', 'hello world!', version=2)
# Get the default version (assuming version=1)
>>> cache.get('my_key')
None
# Get version 2 of the same key
>>> cache.get('my_key', version=2)
'hello world!'
```

特定键的版本可以使用 `incr_version()` 和 `decr_version()` 方法递增和递减。这使得特定键可以被碰撞到新版本，而其他键不受影响。继续我们前面的例子：

```
# Increment the version of 'my_key'
>>> cache.incr_version('my_key')
# The default version still isn't available
>>> cache.get('my_key')
None
# Version 2 isn't available, either
>>> cache.get('my_key', version=2)
None
# But version 3 *is* available
>>> cache.get('my_key', version=3)
'hello world!'
```

缓存键转换

如前两节所述，用户提供的缓存键不是逐字使用的 - 它与缓存前缀和键版本结合以提供最终缓存键。默认情况下，使用冒号连接这三个部分以生成最终字符串：

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), key])
```

如果要以不同的方式组合这些部分，或对最终密钥应用其他处理（例如，获取关键部分的哈希摘要），则可以提供自定义键功能。

`KEY_FUNCTION` 缓存设置指定与上述 `make_key()` 的原型匹配的函数的虚线路径。如果提供，将使用此自定义键功能，而不是默认键组合功能。

缓存关键警告

Memcached是最常用的生产缓存后端，它不允许超过250个字符或包含空格或控制字符的缓存键，并且使用这样的键将导致异常。为了鼓励缓存可移植代码并最小化令人不快的惊喜，其他内置缓存后端会发出警告（ `django.core.cache.backends.base.CacheKeyWarning` ），如果使用一个键， memcached上的错误。

If you are using a production backend that can accept a wider range of keys (a custom backend, or one of the non-memcached built-in backends), and want to use this wider range without warnings, you can silence `CacheKeyWarning` with this code in the `management` module of one of your `INSTALLED_APPS` :

```
import warnings

from django.core.cache import CacheKeyWarning

warnings.simplefilter("ignore", CacheKeyWarning)
```

如果您要为其中一个内置后端提供自定义键验证逻辑，则可以对其进行子类化，仅覆盖 `validate_key` 方法，并按照的说明使用自定义缓存后端。例如，要为 `locmem` 后端执行此操作，请将此代码放在模块中：

```
from django.core.cache.backends.locmem import LocMemCache

class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        """Custom validation, raising exceptions or warnings as needed."""
        # ...
```

...并在 `CACHES` 设置的 `BACKEND` 部分使用此类别的Python路径。

下游缓存

到目前为止，本文档专注于缓存您的自己的数据。但是另一种类型的缓存也与Web开发相关：由“下游”缓存执行的缓存。这些是在请求到达您的网站之前为用户缓存页面的系统。

下面是一些下游缓存的例子：

- 您的ISP可能会缓存某些网页，因此如果您从http://example.com/请求了某个网页，您的ISP会向您发送该网页，而无需直接访问example.com。

example.com的维护者不知道这个缓存；该ISP位于example.com和您的Web浏览器之间，透明地处理所有的缓存。
- 您的Django网站可能位于代理缓存后面，例如Squid Web代理缓存（http://www.squid-cache.org/），缓存页面性能。在这种情况下，每个请求将首先由代理处理，并且只有在需要时才会传递给您的应用程序。
- 您的Web浏览器也缓存页面。如果网页发出相应的标头，您的浏览器将使用本地缓存副本来处理该页面的后续请求，甚至无需再次联系网页以查看其是否已更改。

下游缓存是一个很好的效率提升，但是有一个危险：许多网页的内容基于身份验证和大量其他变量，缓存系统盲目保存纯粹基于URL的网页可能会暴露不正确或敏感数据到后续这些页面的访问者。

例如，假设您操作Web电子邮件系统，并且“收件箱”页面的内容显然取决于哪个用户登录。如果ISP盲目缓存您的网站，则通过该ISP登录的第一个用户将为其后的访问者缓存其用户特定的收件箱页面。这不酷。

幸运的是，HTTP提供了这个问题的解决方案。存在多个HTTP报头以指示下游高速缓存根据指定的变量来使其高速缓存内容不同，并且指示高速缓存机制不缓存特定页面。我们将在接下来的部分中看一些这些标题。

使用Vary头

`Vary` 头定义了缓存机制在构建其缓存键时应考虑哪些请求头。例如，如果网页的内容取决于用户的语言偏好，则该页被称为“根据语言而变化”。

默认情况下，Django的缓存系统使用请求的完全限定网址（例如，`"http://www.example.com/stories/2005/?order_by=author"`）创建其缓存密钥。这意味着对该网址的每个请求都将使用相同的缓存版本，而不考虑用户代理的差异（例如Cookie或语言首选项）。但是，如果此网页根据请求标头（例如Cookie，语言或用户代理）的某些差异产生不同的内容，则需要使用 `Vary` 标头来指示缓存页面输出依赖于这些东西的机制。

Changed in Django 1.7:

缓存键使用请求的完全限定URL，而不仅仅是路径和查询字符串。

要在Django中执行此操作，请使用方便的 `django.views.decorators.vary.vary_on_headers()` 视图装饰器，如下所示：

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def my_view(request):
    # ...
```

在这种情况下，缓存机制（例如Django自己的缓存中间件）将缓存每个唯一用户代理的页面的单独版本。

使用 `vary_on_headers` 装饰器而不是手动设置 `Vary` 标头的优点（使用类似 `response ['Vary'] = 'user-agent'` ）是装饰器将添加到 `Vary`

您可以将多个标头传送至 `vary_on_headers()` ：

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    # ...
```

这告诉下游缓存在_两者上变化，这意味着用户代理和cookie的每个组合将获得自己的缓存值。_For example, a request with the user-agent `Mozilla` and the cookie value `foo=bar` will be considered different from a request with the user-agent `Mozilla` and the cookie value `foo=ham` .

因为cookie的变化很常见，所以有一个 `django.views.decorators.vary.vary_on_cookie()` 装饰器。这两个视图是等效的：

```
@vary_on_cookie
def my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(request):
    # ...
```

您传递给 `vary_on_headers` 的标头不区分大小写； `"User-Agent"` 与 `"user-agent"` 相同。

您也可以直接使用辅助函数 `django.utils.cache.patch_vary_headers()` 。此函数设置或添加到 `Vary 标题` 。例如：

```
from django.utils.cache import patch_vary_headers

def my_view(request):
    # ...
    response = render_to_response('template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

`patch_vary_headers` 采用 `HttpResponse` 实例作为其第一个参数，以及不区分大小写的标题名称的列表/元组作为其第二个参数。

有关Vary头的更多信息，请参阅官方Vary spec。

控制缓存：使用其他头

缓存的其他问题是数据的隐私和数据应该存储在级联的缓存中的位置的问题。

用户通常面临两种缓存：它们自己的浏览器缓存（私有缓存）和它们的提供者的缓存（公共缓存）。公共缓存由多个用户使用并由其他人控制。这会导致敏感数据出现问题，例如您的银行帐号存储在公共缓存中。因此，Web应用程序需要一种方法来告诉缓存哪些数据是私有的，哪些是公共的。

解决方案是指示页面的缓存应为“私有”。要在Django中执行此操作，请使用 `cache_control` 视图装饰器。例：

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    # ...
```

这个装饰器负责在幕后发送适当的HTTP标头。

注意，缓存控制设置“private”和“public”是互斥的。装饰器确保“public”指令被删除，如果“private”应该被设置（反之亦然）。两个指令的示例使用将是提供私人和公共条目的博客站点。公共条目可以在任何共享缓存上缓存。以下代码使用 `django.utils.cache.patch_cache_control()` ，手动方式修改缓存控件头（它由 `cache_control` 装饰器内部调用）：

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous():
        response = render_only_public_entries()
        patch_cache_control(response, public=True)
    else:
        response = render_private_and_public_entries(request.use
r)
        patch_cache_control(response, private=True)

    return response
```

还有一些其他方法来控制缓存参数。例如，HTTP允许应用程序执行以下操作：

- 定义页面应缓存的最大时间。
- 指定缓存是否应始终检查较新版本，仅在没有更改时传递缓存的内容。（即使服务器页面更改，一些缓存可能会提供缓存内容，因为缓存副本尚未到期。）

在Django中，使用 `cache_control` 视图装饰器来指定这些缓存参数。在此示例中， `cache_control` 告诉缓存在每次访问时重新验证缓存，并将缓存的版本最多保存3,600秒：

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    # ...
```

任何有效的 `Cache-Control` HTTP指令在 `cache_control()` 中有效。这里有一个完整的列表：

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`
- `proxy_revalidate=True`
- `max_age=num_seconds`
- `s_maxage=num_seconds`

有关缓存控制HTTP指令的说明，请参见缓存控制规范。

（请注意，缓存中间件已经将缓存头的max-age设置为 `CACHE_MIDDLEWARE_SECONDS` 设置的值。如果您在 `cache_control` 装饰器中使用自定义 `max_age` ，装饰器将优先，并且标头值将正确合并。）

If you want to use headers to disable caching altogether, `django.views.decorators.cache.never_cache` is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. 例：

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    # ...
```

MIDDLEWARE_CLASSES的顺序

如果您使用缓存中间件，请务必将每一半放在 `MIDDLEWARE_CLASSES` 设置中的正确位置。这是因为缓存中间件需要知道通过哪个头来改变缓存存储。中间件总是向 `Vary` 响应标头添加一些东西。

`UpdateCacheMiddleware` 在响应阶段运行，其中中间件按相反顺序运行，因此列表顶部的项目在响应阶段运行最后。因此，您需要确保 `UpdateCacheMiddleware` 在之前出现可能会为 `Vary` 标题添加内容的任何其他中间件。以下中间件模块：

- `SessionMiddleware` 添加 `Cookie`
- `GZipMiddleware` 添加 `Accept-Encoding`
- `LocaleMiddleware` 添加 `Accept-Language`

`FetchFromCacheMiddleware` , on the other hand, runs during the request phase, where middleware is applied first-to-last, so an item at the top of the list runs *first* during the request phase. `FetchFromCacheMiddleware` 也需要在其他中间件更新 `Vary` 标题后运行，因此 `FetchFromCacheMiddleware` 必须在之后这样做。

日志

日志快速入门

Django 使用Python 内建的 `logging` 模块打印日志。该模块的用法在Python 本身的文档中有详细的讨论。如果你从来没有使用过Python 的logging 框架（或者即使使用过），请参见下面的快速导论。

logging 的组成

Python 的logging 配置由四个部分组成：

- Loggers
- Handlers
- Filters
- Formatters

Loggers

Logger 为日志系统的入口。每个logger 是一个具名的容器，可以向它写入需要处理的消息。

每个logger 都有一个日志级别。日志级别表示该logger 将要处理的消息的严重性。Python 定义以下几种日志级别：

- `DEBUG` ：用于调试目的的底层系统信息
- `INFO` ：普通的系统信息
- `WARNING` ：表示出现一个较小的问题。
- `ERROR` ：表示出现一个较大的问题。
- `CRITICAL` ：表示出现一个致命的问题。

写入logger 的每条消息都是一个日志记录。每个日志记录也具有一个日志级别，它表示对应的消息的严重性。每个日志记录还可以包含描述正在打印的事件的有用元信息。这些元信息可以包含很多细节，例如回溯栈或错误码。

当给一条消息给logger 时，会将消息的日志级别与logger 的日志级别进行比较。如果消息的日志级别大于等于logger 的日志级别，该消息将会往下继续处理。如果小于，该消息将被忽略。

Logger 一旦决定消息需要处理，它将传递该消息给一个*Handler*。

Handlers

Handler 决定如何处理logger 中的每条消息。它表示一个特定的日志行为，例如将消息写到屏幕上、写到文件中或者写到网络socket。

与logger 一样，handler 也有一个日志级别。如果消息的日志级别小于handler 的级别，handler 将忽略该消息。

Logger 可以有多个handler，而每个handler 可以有不同的日志级别。利用这种方式，可以根据消息的重要性提供不同形式的处理。例如，你可以用一个handler 将 `ERROR` 和 `CRITICAL` 消息发送给一个页面服务，而用另外一个hander 将所有的消息（包括 `ERROR` 和 `CRITICAL` 消息）记录到一个文件中用于以后进行分析。

Filters

Filter 用于对从logger 传递给handler 的日志记录进行额外的控制。

默认情况下，满足日志级别的任何消息都将被处理。通过安装一个filter，你可以对日志处理添加额外的条件。例如，你可以安装一个filter，只允许处理来自特定源的 `ERROR` 消息。

Filters 还可以用于修改将要处理的日志记录的优先级。例如，如果日志记录满足特定的条件，你可以编写一个filter 将日志记录从 `ERROR` 降为 `WARNING` 。

Filters 可以安装在logger 上或者handler 上；多个filter 可以串联起来实现多层filter 行为。

Formatters

最后，日志记录需要转换成文本。Formatter 表示文本的格式。Fomatter 通常由包含日志记录属性的Python 格式字符串组成；你也可以编写自定义的fomatter 来实现自己的格式。

使用logging

配置好logger、handler、filter 和formatter 之后，你需要在代码中放入logging 调用。使用logging 框架非常简单。下面是个例子：

```
# import the logging library
import logging

# Get an instance of a logger
logger = logging.getLogger(__name__)

def my_view(request, arg1, arg):
    ...
    if bad_mojo:
        # Log an error message
        logger.error('Something went wrong!')
```

就是这样！每次满足 `bad_mojo` 条件，将写入一条错误日志记录。

命名logger

`logging.getLogger()` 调用获取（如有必要则创建）一个logger 的实例。Logger 实例通过名字标识。Logger 使用名称的目的是用于标识其配置。

Logger 的名称习惯上通常使用 `__name__` ，即包含该logger 的Python 模块的名字。这允许你基于模块filter 和handle 日志调用。如果你想使用其它方式组织日志消息，可以提供点号分隔的名称来标识你的logger：

```
# Get an instance of a specific named logger
logger = logging.getLogger('project.interesting.stuff')
```

点号分隔的logger 名称定义一个层级。 `project.interesting` logger 被认为是 `project.interesting.stuff` logger 的上一级； `project` logger 是 `project.interesting` logger 的上一级。

层级为何如此重要？因为可以设置logger 传播它们的logging 调用给它们的上一级。利用这种方式，你可以在根logger 上定义一系列的handler，并捕获子logger 中的所有logging 调用。在 `project` 命名空间中定义的handler 将捕获 `project.interesting` 和 `project.interesting.stuff` logger 上的所有日志消息。

这种传播行为可以基于每个logger 进行控制。如果你不想让某个logger 传播消息给它的上一级，你可以关闭这个行为。

logging 调用

Logger 实例为每个默认的日志级别提供一个入口方法：

- `logger.debug()`
- `logger.info()`
- `logger.warning()`
- `logger.error()`
- `logger.critical()`

还有另外两个调用：

- `logger.log()` ：打印消息时手工指定日志级别。
- `logger.exception()` ：创建一个 `ERROR` 级别日志消息，它封装当前异常栈的帧。

配置logging

当然，只是将logging 调用放入你的代码中还是不够的。你还需要配置logger、handler、filter 和formatter 来确保日志的输出是有意义的。

Python 的logging 库提供几种配置logging 的技术，从程序接口到配置文件。默认情况下，Django 使用dictConfig 格式。

为了配置logging，你需要使用 `LOGGING` 来定义字典形式的logging 设置。这些设置描述你的logging 设置的logger、handler、filter 和formatter，以及它们的日志等级和其它属性。

默认情况下， `LOGGING` 设置与*Django 的默认logging 配置*进行合并。

如果 `LOGGING` 中的 `disable_existing_loggers` 键为 `True` （默认值），那么默认配置中的所有logger 都将禁用。Logger 的禁用与删除不同；logger 仍然存在，但是将默默丢弃任何传递给它的信息，也不会传播给上一级logger。所以，你应该非常小心使用 `'disable_existing_loggers': True` ；它可能不是你想要的。你可以设置 `disable_existing_loggers` 为 `False` ，并重新定义部分或所有的默认loggers；或者你可以设置 `LOGGING_CONFIG` 为 `None` ，并 自己处理 *logging* 配置。

Logging 的配置属于Django `setup()` 函数的一部分。所以，你可以肯定在你的项目代码中logger 是永远可用的。

示例

dictConfig 格式的完整文档是logging 字典配置最好的信息源。但是为了让你尝尝，下面是几个例子。

首先，下面是一个简单的配置，它将来自*django.request* logger 的所有日志请求写入到一个本地文件：

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': '/path/to/django/debug.log',
        },
    },
    'loggers': {
        'django.request': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

如果你使用这个示例，请确保修改 `'filename'` 路径为运行Django 应用的用户有权限写入的一个位置。

其次，下面这个示例演示如何让日志系统将Django 的日志打印到控制台。 `django.request` 和 `django.security` 不会传播日志给上一级。它在本地开发期间可能有用。

默认情况下，这个配置只会将 `INFO` 和更高级别的日志发送到控制台。Django 中这样的日志信息不多。可以设置环境变量 `DJANGO_LOG_LEVEL=DEBUG` 来看看Django 的debug 日志，它包含所有的数据库查询所以非常详尽。

```
import os

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'level': os.getenv('DJANGO_LOG_LEVEL', 'INFO'),
        },
    },
}
```

最后，下面是相当复杂的一个logging 设置：

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(pr
ocess)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level': 'DEBUG',
            'class': 'logging.NullHandler',
```

```
        },
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers': ['null'],
            'propagate': True,
            'level': 'INFO',
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

这个logging 配置完成以下事情：

- 以'dictConfig version 1'格式解析配置。目前为止，这是dictConfig 格式唯一的版本。
- 定义两个formatter：
    - `simple`，它只输出日志的级别（例如，`DEBUG`）和日志消息。

      `format` 字符串是一个普通的Python 格式化字符串，描述每行日志的细节。输出的完整细节可以在formatter 文档中找到。
    - `verbose`，它输出日志级别、日志消息，以及时间、进程、线程和生成日志消息的模块。
- 定义filter —— `project.logging.SpecialFilter`，并使用别名 `special`。如果filter 在构造时要求额外的参数，可以在filter 的配置字段中用额外的键提供。在这个例子中，在实例化 `SpecialFilter` 时，`foo` 参数的值将使用 `bar`。
- 定义三个handler：

    - `null` ，一个NullHandler，它传递 `DEBUG` （和更高级）的消息给 `/dev/null` 。
    - `console` ，一个StreamHandler，它将打印 `DEBUG` （和更高级）的消息到stderr。这个handler 使用 `simple` 输出格式。
    - `mail_admins` ，一个AdminEmailHandler，它将用邮件发送 `ERROR` （和更高级）的消息到站点管理员。这个handler 使用 `special` filter。
- 配置三个logger：
    - `django` ，它传递所有 `INFO` 和更高级的消息给 `null` handler。
    - `django.request` ，它传递所有 `ERROR` 消息给 `mail_admins` handler。另外，标记这个logger 不 向上传播消息。这表示写入 `django.request` 的日志信息将不会被 `django` logger 处理。
    - `myproject.custom` ，它传递所有 `INFO` 和更高级的消息并通过 `special` filter 的消息给两个handler —— `console` 和 `mail_admins` 。这表示所有 `INFO` （和更高级）的消息将打印到控制台上； `ERROR` 和 `CRITICAL` 消息还会通过邮件发送出来。

自定义logging 配置

如果你不想使用Python 的dictConfig 格式配置logger，你可以指定你自己的配置模式。

`LOGGING_CONFIG` 设置定义一个可调用对象，将它用来配置Django 的logger。默认情况下，它指向Python 的 `logging.config.dictConfig()` 函数。但是，如果你想使用不同的配置过程，你可以使用其它只接受一个参数的可调用对象。配置logging 时，将使用 `LOGGING` 的内容作为参数的值。

禁用logging 配置

如果你完全不想配置logging（或者你想使用自己的方法手工配置logging），你可以设置 `LOGGING_CONFIG` 为 `None` 。这将禁用*Django 默认logging* 的配置过程。下面的示例禁用Django 的logging 配置，然后手工配置logging：

settings.py

```
LOGGING_CONFIG = None

import logging.config
logging.config.dictConfig(...)
```

设置 `LOGGING_CONFIG` 为 `None` 只表示禁用自动配置过程，而不是禁用logging 本身。如果你禁用配置过程，Django 仍然执行logging 调用，只是调用的是默认定义的logging 行为。

Django’s logging extensions

Django 提供许多工具用于处理在网站服务器环境中独特的日志需求。

Loggers

Django 提供几个内建的logger。

django

`django` 是一个捕获所有信息的logger。消息不会直接提交给这个logger。

django.request

记录与处理请求相关的消息。5XX 响应作为 `ERROR` 消息；4XX 响应作为 `WARNING` 消息。

这个logger 的消息具有以下额外的上下文：

- `status_code` ：请求的HTTP 响应码。
- `request` ：生成日志信息的请求对象。

django.db.backends

与数据库交互的代码相关的消息。例如，HTTP请求执行应用级别的SQL 语句将以 `DEBUG` 级别记录到该logger。

这个logger 的消息具有以下额外的上下文：

- `duration` ：执行SQL 语句花费的时间。
- `sql` ：执行的SQL 语句。
- `params` ：SQL 调用中用到的参数。

由于性能原因，SQL的日志只在 `设置` 之后开启。DEBUG 设置为 `True` ，无论日志级别或者安装的处理器是什么。

这里的日志不包含框架级别的的初始化（例如， `SET TIMEZONE` ）和事务管理查询（例如， `BEGIN` 、 `COMMIT` 和 `ROLLBACK` ）。如果你希望看到所有的数据库查询，可以打开数据库中的查询日志。

django.security.*

Security logger 将收到任何出现 `SuspiciousOperation` 的消息。SuspiciousOperation 的每个子类型都有一个子logger。日志的级别取决于异常处理的位置。大部分情况是一个warning 日志，而如果 `SuspiciousOperation` 到达

WSGI handler 则记录为一个error。例如，如果请求中包含的HTTP `Host` 头部与 `ALLOWED_HOSTS` 不匹配，Django 将返回400 响应，同时将记录一个error 消息到 `django.security.DisallowedHost` logger。

默认情况下只会配置 `django.security` logger，其它所有的子logger 都将传播给上一级logger。 `django.security` logger 的配置与 `django.request` logger 相同，任何error 消息将用邮件发送给站点管理员。由于 `SuspiciousOperation` 导致400 响应的请求不会在 `django.request` logger 中记录日志，而只在 `django.security` logger 中记录日志。

若要默默丢弃某种类型的SuspiciousOperation，你可以按照下面的示例覆盖其logger：

```
'loggers': {
    'django.security.DisallowedHost': {
        'handlers': ['null'],
        'propagate': False,
    },
},
```

django.db.backends.schema

New in Django 1.7.

当迁移框架执行的SQL 查询会改变数据库的模式时，则记录这些SQL 查询。注意，它不会记录 `RunPython` 执行的查询。

Handlers

在Python logging 模块提供的handler 基础之上，Django 还提供另外一个handler。

class `AdminEmailHandler`(*include_html=False*, *email_backend=None*)[source]

这个handler 将它收到的每个日志信息用邮件发送给站点管理员。

如果日志记录包含 `request` 属性，该请求的完整细节都将包含在邮件中。

如果日志记录包含栈回溯信息，该栈回溯也将包含在邮件中。

`AdminEmailHandler` 的 `include_html` 参数用于控制邮件中是否包含HTML 附件，这个附件包含 `DEBUG` 为 `True` 时的完整网页。若要在配置中设置这个值，可以将它包含在 `django.utils.log.AdminEmailHandler` handler 的定义中，像下面这样：

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
    }
},
```

注意，邮件中的HTML 包含完整的回溯栈，包括栈每个层级局部变量的名称和值以及你的Django 设置。这些信息可能非常敏感，你也许不想通过邮件发送它们。此时可以考虑使用类似Sentry 这样的东西，回溯栈的完整信息和安全信息不会 通过邮件发送。你还可以从错误报告中显式过滤掉特定的敏感信息 —— 更多信息参见过滤错误报告。

通过设置 `AdminEmailHandler` 的 `email_backend` 参数，可以覆盖handler 使用的*email backend*，像这样：

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'email_backend': 'django.core.mail.backends.filebased.Em
ailBackend',
    }
},
```

默认情况下，将使用 `EMAIL_BACKEND` 中指定的邮件后端。

`send_mail` (*subject*, *message*, **args*, ***kwargs*)[source]

New in Django 1.8.

发送邮件给管理员用户。若要自定它的行为，可以子类化 `AdminEmailHandler` 类并覆盖这个方法。

Filters

在Python logging 模块提供的过滤器的基础之上，Django 还提供两个过滤器。

class `CallbackFilter` (*callback*)[source]

这个过滤器接受一个回调函数（它接受一个单一参数，也就是要记录的东西），并且对每个传递给过滤器的记录调用它。如果回调函数返回False，将不会进行记录的处理。

例如，要从admin邮件中过滤掉 `UnreadablePostError` （只在用户取消上传时产生），你可以创建一个过滤器函数：

```
from django.http import UnreadablePostError

def skip_unreadable_post(record):
    if record.exc_info:
        exc_type, exc_value = record.exc_info[:2]
        if isinstance(exc_value, UnreadablePostError):
            return False
    return True
```

然后把它添加到logger的配置中：

```
'filters': {
    'skip_unreadable_posts': {
        '()': 'django.utils.log.CallbackFilter',
        'callback': skip_unreadable_post,
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['skip_unreadable_posts'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class `RequireDebugFalse` [source]

这个过滤器只在设置后传递记录。DEBUG 为 False。

这个过滤器遵循 `LOGGING` 默认的配置，以确保 `AdminEmailHandler` 只在 `DEBUG` 为 `False` 的时候发送错误邮件。

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse',
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class `RequireDebugTrue` [source]

这个过滤器类似于`RequireDebugFalse`，除了记录只在 `DEBUG` 为 `True` 时传递的情况。

Django's default logging configuration

默认情况下，Django 的logging 配置如下：

当 `DEBUG` 为 `True` 时：

- `django` 的全局logger会向控制台发送级别等于或高级 `INFO` 的所有消息。Django在这个时候并不会做任何日志调用（所有在 `DEBUG` 级别上的日志，或者被 `django.request` 和 `django.security` 处理的日志）。
- `py.warnings` logger，它处理来自 `warnings.warn()` 的消息，会向控制台发送消息。

当 `DEBUG` 为 `False` 时：

- `django.request` 和 `django.security` loggers 向 `AdminEmailHandler` 发送带有 `ERROR` 或 `CRITICAL` 级别的消息。这些logger 会忽略任何级别等于或小于 `WARNING` 的信息，被记录的日志不会传递给其他logger（它们不会传递给 `django` 的全局 logger，即使 `DEBUG` 为 `True` ）。

另见配置日志来了解如何补充或者替换默认的日志配置。

> 译者：Django 文档协作翻译小组，原文：Logging。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

发送邮件

尽管Python 通过 `smtplib` 模块使得发送邮件很简单，Django 仍然在此基础上提供了几个轻量的封装包。这些封装包使得发送邮件非常快速、让开发中测试发送邮件变得很简单、并且支持不使用SMTP 的平台。

这些代码包含在 `django.core.mail` 模块中。

简单例子

两行代码实现：

```
from django.core.mail import send_mail

send_mail('Subject here', 'Here is the message.', 'from@example.com',
    ['to@example.com'], fail_silently=False)
```

邮件使用 `EMAIL_HOST` 和 `EMAIL_PORT` 设置指定的SMTP 主机和端口发送。如果settings中设置了 `EMAIL_HOST_USER` 和 `EMAIL_HOST_PASSWORD` 它们将被用来验证SMTP主机，并且如果设置了 `EMAIL_USE_TLS` 和 `EMAIL_USE_SSL` 它们将控制是否使用相应的加密链接。

注

`django.core.mail` 发送邮件时使用的字符集将按照你在settings中的 `DEFAULT_CHARSET` 项来设置。

发送邮件()

`send_mail` (*subject*, *message*, *from_email*, *recipient_list*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*, *html_message=None*)[source]

发送邮件最简单的方法是使用 `django.core.mail.send_mail()` 。

`subject` 、 `message` 、 `from_email` 和 `recipient_list` 参数是必须的。

- `subject` ：一个字符串。
- `message` ：一个字符串。
- `from_email` ：一个字符串。
- `recipient_list` ：一个由邮箱地址组成的字符串列表。 `recipient_list` 中的每一个成员都会在邮件信息的“To:”区域看到其它成员。
- `fail_silently` ：一个布尔值。如果设置为 `False` ， `send_mail` 将引发一

个 `smtplib.` SMTPException异常. 查看 `smtplib` 文档中列车出的所有可能的异常， 它们都是 `SMTPException` 的子类。

- `auth_user`：可选的用户名用来验证SMTP服务器。如果没有提供这个值，Django 将会使用settings中 `EMAIL_HOST_USER` 的值。
- `auth_password`：可选的密码用来验证 SMTP 服务器。如果没有提供这个值， Django 将会使用settings中 `EMAIL_HOST_PASSWORD` 的值。
- `connection`：可选的用来发送邮件的电子邮件后端。如果没有指定，将使用缺省的后端实例。查看 *Email backends* 文档来获取更多细节。
- `html_message`：如果提供了 `html_message` ，会导致邮件变成 *multipart/alternative* ， `message` 格式变成 *text/plain* ， `html_message` 格式变成 *text/html* 。

返回值将是成功传递的消息的数量（可以是 `0` 或 `1` ，因为它只能发送一个消息）。

New in Django 1.7:

已添加 `html_message` 参数。

send_mass_mail()

`send_mass_mail` (*datatuple*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*)[source]

`django.core.mail.send_mass_mail()` 用来处理大批量邮件任务

`datatuple` 是一个元组，其中元素的格式如下

```
(subject, message, from_email, recipient_list)
```

`fail_silently` , `auth_user` , `auth_password` 与 `send_mail()` 中的方法相同.

`datatuple` 的每个单独元素产生单独的电子邮件。在 `send_mail()` 中，同一 `recipient_list` 中的收件人将看到电子邮件的“收件人：”字段中的其他地址。

例如，以下代码将向两个不同的收件人集发送两个不同的消息；然而，只有一个到邮件服务器的连接将被打开：

```
message1 = ('Subject here', 'Here is the message', 'from@example
.com', ['first@example.com', 'other@example.com'])
message2 = ('Another Subject', 'Here is another message', 'from@
example.com', ['second@test.com'])
send_mass_mail((message1, message2), fail_silently=False)
```

返回值将是已成功传递的消息数。

send_mass_mail()vs. send_mail()

`send_mass_mail()` 和 `send_mail()` 的主要差别是 `send_mail()` 每次运行时打开一个到邮箱服务器的连接，而 `send_mass_mail()` 对于所有的信息都只使用一个连接。这使得 `send_mass_mail()` 更高效点.

mail_admins()

`mail_admins` (*subject*, *message*, *fail_silently=False*, *connection=None*, *html_message=None*)[source]

`django.core.mail.mail_admins()` 是向 `ADMINS` 设置中定义的网站管理员发送电子邮件的快捷方式。

`mail_admins()` 以 `EMAIL_SUBJECT_PREFIX` 设置的值为主题添加前缀，即 `“[Django] t7>` 。

电子邮件的“From：”标头将是 `SERVER_EMAIL` 设置的值。

此方法的存在是为了方便和可读性。

如果提供 `html_message` ，则生成的电子邮件将是具有 `message` 作为*text / plain* 的*multipart* /内容类型和 *`html_message`* 作为_*text / html*内容类型。_

mail_managers()

`mail_managers` (*subject*, *message*, *fail_silently=False*, *connection=None*, *html_message=None*)[source]

`django.core.mail.mail_managers()` is just like `mail_admins()` , except it sends an email to the site managers, as defined in the `MANAGERS` setting.

例子

这会向john @示例发送一封电子邮件。com和jane @示例。com，他们都出现在“To：”：

```
send_mail('Subject', 'Message.', 'from@example.com',
    ['john@example.com', 'jane@example.com'])
```

这会向john @示例发送消息。com和jane @示例。com，他们都收到一个单独的电子邮件：

```
datatuple = (
    ('Subject', 'Message.', 'from@example.com', ['john@example.com']),
    ('Subject', 'Message.', 'from@example.com', ['jane@example.com']),
)
send_mass_mail(datatuple)
```

防止标题注入

Header injection是一个安全漏洞，攻击者插入额外的电子邮件标题控制”TO：“和”FROM：“在你的脚本生成的电子邮件

Django的电子邮件上述的功能概述都是通过标头值禁止换行来防止头注入。 如果任意的 `subject` , `from_email` 或者 `recipient_list` 包含一个新行 (in either Unix, Windows or Mac style), 则email功能函数 (e.g. `send_mail()`) 将会引起 `django.core.mail.BadHeaderError` (`ValueError` 的子类) 因此，并不会发送邮件。你有责任在把数据发送到email功能函数之前进行验证。

如果一条 `message` 字符串包含了一个header开头，这个header将会被简单的打印为这条email的第一个字节。

这里有个简单的例子， `subject` , `message` 和 `from_email` 来自于 request's POST 数据, 发送数据到 admin@example.com并在完成后重定向到“/contact/thanks/” ：

```
from django.core.mail import send_mail, BadHeaderError
from django.http import HttpResponse, HttpResponseRedirect

def send_email(request):
    subject = request.POST.get('subject', '')
    message = request.POST.get('message', '')
    from_email = request.POST.get('from_email', '')
    if subject and message and from_email:
        try:
            send_mail(subject, message, from_email, ['admin@example.com'])
        except BadHeaderError:
            return HttpResponse('Invalid header found.')
        return HttpResponseRedirect('/contact/thanks/')
    else:
        # In reality we'd use a form class
        # to get proper validation errors.
        return HttpResponse('Make sure all fields are entered and valid.')
```

EmailMessage类

Django的 `send_mail()` 和 `send_mass_mail()` 是利用了 `EmailMessage` class.的简单封装。

并不是所有的 `EmailMessage` 类 的特性都可以通过 `send_mail()` 和相关的封装函数来实现的。如果你想用更高级的特性， 比如 BCC'ed recipients, 附件上传, 多部分邮件, 你需要直接创建 `EmailMessage` 实例。

注意

这是一个设计特点。 `send_mail()` 和相关的函数是django最初提供的接口。然而，.参数列表随着时间的推移不断增长。转向面向对象设计的邮件信息，并且保留最初的功能以向后兼任是明智之举。

`EmailMessage` 负责创建电子邮件本身。 *email backend* 才是负责发送email的。

方便起见， `EmailMessage` 提供了一个简单的 `send()` 方法来发送单一邮件。. 如果你需要发送多份邮件, email的后端 API *provides an alternative*.

EmailMessage对象

class `EmailMessage`

`EmailMessage` 这个类由一下这些参数来实例化 (in the given order, if positional arguments are used). 所有参数都是可选的并且可以在 `send()` 方法之前的任意时间设置。

- `subject` ：电子邮件的主题行。
- `body` ：正文文本。这应该是纯文本消息。
- `from_email` ：发件人的地址。表单合法， `fred@example.com` 和 `Fred &lt； fred@example.com&gt；` 如果省略，则使用 `DEFAULT_FROM_EMAIL` 设置。
- `to` ：收件人地址的列表或元组。
- `bcc` ：发送电子邮件时在"密件抄送"标头中使用的地址列表或元组。
- `connection` ：电子邮件后端实例。如果要对多个消息使用相同的连接，请使用此参数。如果省略，则在调用 `send()` 时创建新连接。
- `attachments` ：要放在邮件上的附件列表。这些可以是 `email.` MIMEBase.MIMEBase实例或 `（文件名， 内容， mimetype）`
- `headers` ：一个额外标题的字典放在消息上。键是标题名称，值是标题值。它取决于调用者确保电子邮件消息的头名称和值的格式正确。相应的属性为 `extra_headers` 。
- `cc` ：发送电子邮件时在"Cc"标头中使用的收件人地址的列表或元组。
- `reply_to` ：发送电子邮件时在"回复"标题中使用的收件人地址的列表或元组。

Changed in Django 1.8:

已添加 `reply_to` 参数。

例如：

```
email = EmailMessage('Hello', 'Body goes here', 'from@example.co
m',
            ['to1@example.com', 'to2@example.com'], ['bcc@exampl
e.com'],
            reply_to=['another@example.com'], headers={'Message-
ID': 'foo'})
```

该类有以下方法：

- `send(fail_silently=False)` 发送消息。如果在构建电子邮件时指定了连接，则将使用该连接。否则，将实例化并使用默认后端的实例。如果关键字参数 `fail_silently` 是 `True` ，则发送消息时抛出的异常将被取消。如果接受者为空，这并不会引起异常！
- `message()` 构造一个 `django.core.mail.SafeMIMEText` 对象（Python的 `email.` MIMEText.MIMEText类）或一个 `django.core.mail.SafeMIMEMultipart` 对象保存要发送的消息。如果您需要扩展 `EmailMessage` 类，那么您可能想要覆盖此方法，将所需的内容放入MIME对象中。
- `recipients()` 返回邮件的所有收件人列表，无论这些收件人是记录在 `to` ， `cc` 或 `bcc` 属性。这是您在子类化时可能需要覆盖的另一种方法，因为在发送消息时，需要告知SMTP服务器收件人的完整列表。如果您添加另一种方式来指定类中的收件人，则还需要从此方法返回。
- `attach()` 创建一个新文件附件并将其添加到消息中。有两种方法调用 `attach()` ：
    - 您可以向其传递一个为 `email.` MIMEBase.MIMEBase实例。这将直接插入到生成的消息中。
    - 或者，您可以传递 `attach()` 三个参数： `filename` ， `content` 和 `mimetype` 。 `filename` 是电子邮件中显示的文件附件的名称， `content` 是附件中包含的数据， `mimetype` 是附件的可选MIME类型。如果你忽略了 `mimetype` , MIME content type将会从你的文件名来猜测。。

      例如：

      ```
      message.attach('design.png', img_data, 'image/png')
      ```

      Changed in Django 1.7:

如果您指定 `message/rfc822` 的 `mimetype` ，它也会接受 `django.core.mail.EmailMessage` 和 `email.message.Message` 。

此外， `message/rfc822` 附件将不再以违反 **RFC 2046**的base64编码，这可能会导致显示附件的问题进化和Thunderbird 。

- `attach_file()` 使用文件系统中的文件创建新附件。使用要附加的文件的路径和可选的用于附件的MIME类型调用它。如果省略MIME类型，则将从文件名中猜出。最简单的用法是：

  ```
  message.attach_file('/images/weather_map.png')
  ```

发送替代内容类型

在你要包含各种乱七八糟的内容在邮件内容时，这非常的有用; 最经典的案例就是发送text和HTML的信息版本。用Django的 email库, 你可以用 `EmailMultiAlternatives` 这个类来完成。这个类是 `EmailMessage` 的子类，它有 `attach_alternative()` 来包含额外的邮件信息版本。所有其他方法（包括类初始化）直接从 `EmailMessage` 继承。

要发送文本和HTML组合，您可以写：

```
from django.core.mail import EmailMultiAlternatives

subject, from_email, to = 'hello', 'from@example.com', 'to@examp
le.com'
text_content = 'This is an important message.'
html_content = '<p>This is an <strong>important</strong> message
.</p>'
msg = EmailMultiAlternatives(subject, text_content, from_email,
[to])
msg.attach_alternative(html_content, "text/html")
msg.send()
```

默认情况下， `EmailMessage` 中 `body` 参数的MIME类型为 `"text/plain"` 。

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html"  # Main content is now text/html
msg.send()
```

邮件后端

邮件的真正发送是通过邮件后端处理的。

邮件后端类具有以下方法：

- `open()` 实例化一个用于发送邮件的长连接。
- `close()` 关闭当前的邮件发送连接。
- `send_messages(email_messages)` 发送一个 `EmailMessage` 对象的列表。如果连接没有打开，该调用将隐式打开连接，并在之后关闭连接。如果连接已经打开，它在邮件发送之后会保持打开状态。

它还可以用作上下文管理器，在需要的时候自动调用 `open()` 和 `close()` ：

```
from django.core import mail

with mail.get_connection() as connection:
    mail.EmailMessage(subject1, body1, from1, [to1],
                      connection=connection).send()
    mail.EmailMessage(subject2, body2, from2, [to2],
                      connection=connection).send()
```

New in Django 1.8:

新增上下文管理器协议。

获取邮件后端的一个实例

位于 `django.core.mail` 中的 `get_connection()` 函数返回邮件后端的一个实例。

`get_connection` (*backend=None*, *fail_silently=False*, **args*, ***kwargs*)[source]

默认情况下，对 `get_connection()` 的调用将返回 `EMAIL_BACKEND` 指定的邮件后端实例。如果你指定 `backend` 参数，则返回相应的后端的一个实例。

`fail_silently` 参数控制后端如何处理错误。如果 `fail_silently` 为True，邮件发送过程中的异常将会默默忽略。

所有其它的参数都直接传递给邮件后端的构造函数。

Django 自带几个邮件发送的后端。除了SMTP 后端（默认），其它后端只用于测试和开发阶段。如果发送邮件有特殊的需求，你可以编写自己的邮件后端。

SMTP 后端

class `backends.smtp.``EmailBackend` ([*host=None*, *port=None*, *username=None*, *password=None*, *use_tls=None*, *fail_silently=False*, *use_ssl=None*, *timeout=None*, *ssl_keyfile=None*, *ssl_certfile=None*, ***kwargs*])

这是默认的后端。邮件将通过SMTP 服务器发送。

每个参数的值如果为 `None` ，则从settings 中获取对应的设置：

- `host` ： `EMAIL_HOST`

- `port` ： `EMAIL_PORT`
- `username` ： `EMAIL_HOST_USER`
- `password` ： `EMAIL_HOST_PASSWORD`
- `use_tls` ： `EMAIL_USE_TLS`
- `use_ssl` ： `EMAIL_USE_SSL`
- `timeout` ： `EMAIL_TIMEOUT`
- `ssl_keyfile` ： `EMAIL_SSL_KEYFILE`
- `ssl_certfile` ： `EMAIL_SSL_CERTFILE`

SMTP 后端是Django 内在的默认配置。如果你想显示指定，可以将下面这行放到settings 中：

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

New in Django 1.7:

添加 `timeout` 参数。如果没有指定，默认的 `timeout` 为 `socket.getdefaulttimeout()` 提供的值，它默认是 `None` （不会超时）。

Changed in Django 1.8:

添加 `ssl_keyfile` 和 `ssl_certfile` 参数以及对应的settings。添加(`EMAIL_TIMEOUT`) 设置以使得可以自定义 `timeout` 。

Console 后端

Console 后端不真正发送邮件，而只是向标准输出打印出邮件。默认情况下，console 后端打印到 `stdout` 。你可以通过在构造connection 时提供 `stream` 参数，来使用一个不同的流对象。

如要指定这个后端，可以将下面这行放入你的settings 中：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

这个后端不能用于生产环境 —— 它只是为了开发方便。

File 后端

File 后端将邮件写到文件中。该后端的每个会话将创建一个新的文件。文件的目录从 `EMAIL_FILE_PATH` 设置或者 `get_connection()` 的 `file_path` 关键字参数获取。

如要指定这个后端，可以将下面这行放入你的settings 中：

```
EMAIL_BACKEND = 'django.core.mail.backends.filebased.EmailBackend'
EMAIL_FILE_PATH = '/tmp/app-messages' # change this to a proper location
```

这个后端不能用于生成环境 —— 它只是为了开发方便。

In-memory 后端

`'locmem'` 后端将邮件保存在 `django.core.mail` 模块的一个特殊属性中。当一封邮件发送时将创建 `outbox` 属性。它是 `EmailMessage` 实例的一个列表，表示每个将要发送的邮件。

如要指定这个后端，可以将下面这行放入你的settings 中：

```
EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```

这个后端不能用于生成环境 —— 它只是为了开发和测试方便。

Dummy 后端

和名字一样，dummy 后端什么也不做。如要指定这个后端，可以将下面这行放入你的settings 中：

```
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'
```

这个后端不能用于生成环境 —— 它只是为了开发方便。

定义自定义电子邮件后端

如果您需要更改电子邮件的发送方式，您可以编写自己的电子邮件后端。您的设置文件中的 `EMAIL_BACKEND` 设置是您的后端类的Python导入路径。

自定义电子邮件后端应该位于 `django.core.mail.backends.base` 模块中的 `BaseEmailBackend` 。自定义电子邮件后端必须实施 `send_messages(email_messages)` 方法。此方法接收 `EmailMessage` 实例的列表，并返回已成功传递的邮件数。如果您的后端有任何持续会话或连接的概念，您还应该实现 `open()` 和 `close()` 方法。请参阅 `smtp.` EmailBackend用于参考实现。

正在发送多封电子邮件

建立和关闭SMTP连接（或任何其他网络连接，这方面）是一个昂贵的过程。如果您有大量电子邮件要发送，则重用SMTP连接是有意义的，而不是每次要发送电子邮件时创建和销毁连接。

有两种方法可以让电子邮件后端重复使用连接。

首先，您可以使用 `send_messages()` 方法。 `send_messages()` 获取 `EmailMessage` 实例（或子类）的列表，并使用单个连接发送它们。

例如，如果您有一个名为 `get_notification_email()` 的函数，该函数返回表示您希望发送的某些周期性电子邮件的 `EmailMessage` 对象列表，则可以使用单次调用send_messages：

```
from django.core import mail
connection = mail.get_connection()   # Use default email connect
ion
messages = get_notification_email()
connection.send_messages(messages)
```

在此示例中，对 `send_messages()` 的调用在后端打开一个连接，发送消息列表，然后再次关闭连接。

第二种方法是使用电子邮件后端上的 `open()` 和 `close()` 方法手动控制连接。 `send_messages()` 将不会手动打开或关闭已打开的连接，因此如果您手动打开连接，则可以控制它何时关闭。例如：

```
from django.core import mail
connection = mail.get_connection()

# Manually open the connection
connection.open()

# Construct an email message that uses the connection
email1 = mail.EmailMessage('Hello', 'Body goes here', 'from@exam
ple.com',
                          ['to1@example.com'], connection=connec
tion)
email1.send() # Send the email

# Construct two more messages
email2 = mail.EmailMessage('Hello', 'Body goes here', 'from@exam
ple.com',
                          ['to2@example.com'])
email3 = mail.EmailMessage('Hello', 'Body goes here', 'from@exam
ple.com',
                          ['to3@example.com'])

# Send the two emails in a single call -
connection.send_messages([email2, email3])
# The connection was already open so send_messages() doesn't clo
se it.
# We need to manually close the connection.
connection.close()
```

配置用于开发的电子邮件

有时候你不想让Django发送电子邮件。例如，在开发网站时，您可能不想发送数千封电子邮件，但您可能需要验证电子邮件是否会在正确的条件下发送给正确的人员，并且这些电子邮件将包含正确的内容。

为本地开发配置电子邮件的最简单方法是使用*console*电子邮件后端。此后端将所有电子邮件重定向到stdout，允许您检查邮件的内容。

*file*电子邮件后端在开发过程中也可以是有用的 - 这个后端将每个SMTP连接的内容转储到可以随时检查的文件。

另一种方法是使用“哑”SMTP服务器，在本地接收电子邮件并将其显示到终端，但实际上不发送任何内容。Python有一个内置的方法来完成这个使用一个单一的命令：

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

此命令将启动一个简单的SMTP服务器侦听localhost的端口1025。此服务器只打印标准输出所有电子邮件标头和电子邮件正文。然后，您只需相应地设置 `EMAIL_HOST` 和 `EMAIL_PORT` 。有关SMTP服务器选项的更详细的讨论，请参阅 `smtpd` 模块的Python文档。

有关在应用程序中单元测试电子邮件发送的信息，请参阅测试文档的*Email services*部分。

Feed聚合框架

Django带有一个高聚合的框架让创造RSS 和 Atom更容易。

创建聚合feed，你要做的仅仅是写一个简短的Python类。你想要创造多少，就能创造多少feeds。

Django 也带有一个低等级的生成feed的API。如果你想要生成一个外部的Web内容或者其他普通的方式，你可以使用它。

高等级的框架

概述

高等级的feed聚合框架由 `Feed` 类提供。新建一个feed，写一个 `Feed` 类，然后指向你的*URLconf*.

Feed 类

一个 `Feed` 类就是一个提供聚合种子的Python类。一个简单的种子（例如新闻的信息种子，或者只展示博客最新消息）更多功能的种子（例如展示博客中允许展示的特定类别的条目）。

Feed类继承自 `django.contrib.syndication.views.Feed` 。它们可以在你代码中的任意一处。

`Feed` 类需要是你*URLconf*中的实例。

一个简单的示例

这个简单的示例，演示了某站点的最近五条新闻的记录：

```
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse
from policebeat.models import NewsItem

class LatestEntriesFeed(Feed):
    title = "Police beat site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to police be
at central."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return item.description

    # item_link is only needed if NewsItem has no get_absolute_u
rl method.
    def item_link(self, item):
        return reverse('news-item', args=[item.pk])
```

配置一个URL到这个feed，在*URLconf*中配置一个入口。例如:

```
from django.conf.urls import url
from myproject.feeds import LatestEntriesFeed

urlpatterns = [
    # ...
    url(r'^latest/feed/$', LatestEntriesFeed()),
    # ...
]
```

注意:

- Feed类继承于 `django.contrib.syndication.views.Feed` .
- `title` , `link` 和 `description` 分别对应 RSS的 `<title>` , `<link>` 和 `<description>` .
- `items()` 就是一个返回包含feed `<item>` 对象列表的方法.当然，这个例子返回 `NewsItem` 对象是使用 Django的 *object-relational mapper*, `items()` 没有返回模型的实例。当然，你可以从Django模型中很容易的获取一些数据， `items()` 可以返回任何你想要的对象。
- 如果你要创建一个Atom feed，而不是RSS feed，你需要使用 `subtitle` 属性替代 `description` 。查看Publishing Atom and RSS feeds in tandem里面的例子。

还有一件事没做。在一个 RSS feed中，每一个 `<item>` 都有一个 `<title>` , `<link>` 和 `<description>` . 我们需要告诉框架那些数据放进这些对象中。

- 在 `<title>` 和 `<description>` 的内容中，Django尝试着在 `Feed` 类中召集 `item_title()` 和 `item_description()` 方法。他们会传入一个自己内部的单独的参数 `item` 。这些都是可选的; 默认的是，unicode 表示的对象都被使用了

  如果你想要做一些特殊的格式化title或者description，*Django templates*可以帮助你。他们的路径会被 `Feed` 类中的 `title_template` 和 `description_template` 参数做特殊处理。会通过模板内容的两个变量来返回每一条记录到模板中：

  - `{% obj %}` ，当前的对象（你在 `items()` 中返回的对象）。
  - `{% site %}` – 一个 `django.contrib.sites.models.Site` 对象代表当前站点。它对 `{% site.domain %}` or `{% site.name %}` 有用。如果你*not*没有Django站点，它会被设置为一个 `RequestSite` 对象。从 *RequestSite section of the sites framework documentation* 获取更多信息。

  从下方的 a complex example 使用一个描述的模板。

  `Feed.``get_context_data` (***kwargs*)

  如果您需要提供超过前面提到的两个变量，还有一种方法可以将附加信息传递到标题和描述模板。您可以在 `Feed` 子类中提供 `get_context_data` 方法的实现。例如:

```
from mysite.models import Article
from django.contrib.syndication.views import Feed

class ArticlesFeed(Feed):
    title = "My articles"
    description_template = "feeds/articles.html"

    def items(self):
        return Article.objects.order_by('-pub_date')[:5]

    def get_context_data(self, **kwargs):
        context = super(ArticlesFeed, self).get_context_data
(**kwargs)
        context['foo'] = 'bar'
        return context
```

  和模板：

```
Something about {{ foo }}: {{ obj.description }}
```

此方法将由 `items()` 返回的列表中的每个项目调用一次，并使用以下关键字参数：

  - `item` ：当前项目。出于向后兼容性原因，此上下文变量的名称为 `{% obj %}` 。
  - `obj` ： `get_object()` 返回的对象。默认情况下，这不会暴露给模板，以避免与 `{% obj %}` ，但您可以在实现 `get_context_data()` 时使用它。
  - `site` ：如上所述的当前网站。
  - `request` ：当前请求。

`get_context_data()` 的行为模仿了*generic views*的行为 - 你应该调用 `super()` 从父类中检索上下文数据，添加您的数据并返回修改后的字典。

- 要指定 `<link>` 的内容，您有两个选项。对于 `items()` 中的每个项目，Django首先尝试调用 `Feed` 类上的 `item_link()` 方法。以类似于标题和描述的方式，它传递单个参数 `item` 。如果该方法不存在，Django尝试对该对象执行 `get_absolute_url()` 方法。 `get_absolute_url()` 和 `item_link()` 应将项目的网址作为普通的Python字符串返回。与 `get_absolute_url()` 一样， `item_link()` 的结果将直接包含在URL中，因此您负责对所有必要的URL进行引用并将其转换为ASCII方法本身。

一个复杂的例子

该框架还通过参数支持更复杂的feed。

例如，网站可以为城市中的每个警察节拍提供最近的犯罪的RSS源。为每个警察节拍创建一个单独的 `Feed` 课；这将违反*DRY principle*并且将数据耦合到编程逻辑。相反，联合框架可让您访问从*URLconf*传递的参数，以便Feed可以根据Feed的网址中的信息输出项目。

警察打击饲料可以通过这样的URL访问：

- `/beats/613/rss/` - 返回613的最近犯罪。
- `/beats/1424/rss/` - 返回1424年最近的犯罪。

这些可以与*URLconf*行匹配，例如：

```
url(r'^beats/(?P<beat_id>[0-9]+)/rss/$', BeatFeed()),
```

与视图类似，URL中的参数与请求对象一起传递到 `get_object()` 方法。

以下是这些特定于节拍的Feed的代码：

```
from django.contrib.syndication.views import FeedDoesNotExist
from django.shortcuts import get_object_or_404

class BeatFeed(Feed):
    description_template = 'feeds/beat_description.html'

    def get_object(self, request, beat_id):
        return get_object_or_404(Beat, pk=beat_id)

    def title(self, obj):
        return "Police beat central: Crimes for beat %s" % obj.b
eat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % ob
j.beat

    def items(self, obj):
        return Crime.objects.filter(beat=obj).order_by('-crime_d
ate')[:30]
```

To generate the feed's <title> , <link> and <description> , Django uses the `title()` , `link()` and `description()` methods. 在前面的示例中，它们是简单的字符串类属性，但是本示例说明它们可以是字符串或方法。对于 `title` ， `link` 和 `description` 中的每一个，Django都遵循此算法：

- 首先，它尝试调用传递 `obj` 参数的方法，其中 `obj` 是由 `get_object()` 返回的对象。
- 如果没有，它试图调用没有参数的方法。
- 如果没有，它使用类属性。

还要注意， `items()` 也遵循相同的算法 - 首先，它尝试 `items(obj)` ，然后 `items()` `items` 类属性（它应该是一个列表）。

我们正在使用模板作为项目描述。它可以很简单：

```
{{ obj.description }}
```

但是，您可以根据需要自由添加格式。

下面的 `ExampleFeed` 类提供了有关 `Feed` 类的方法和属性的完整文档。

指定Feed的类型

默认情况下，此框架中生成的Feed使用RSS 2.0。

要更改此属性，请向您的 Feed 类添加 feed_type 属性，如下所示：

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

请注意，您将 feed_type 设置为类对象，而不是实例。

目前可用的Feed类型有：

- django.utils.feedgenerator.Rss201rev2Feed （RSS 2.01。默认。）
- django.utils.feedgenerator.RssUserland091Feed （RSS 0.91.）
- django.utils.feedgenerator.Atom1Feed （Atom 1.0.）

附件

要指定机柜（例如用于创建Podcast Feed的机柜），请使用 item_enclosure_url ， item_enclosure_length 和 item_enclosure_mime 挂钩。有关用法示例，请参见下面的 ExampleFeed 类。

语言

由联合框架创建的Feed会自动包含适当的 <language> 标记（RSS 2.0）或 xml:lang 属性（Atom）。这直接来自您的 LANGUAGE_CODE 设置。

网址

link 方法/属性可以返回绝对路径（例如 "/blog/" ）或具有完全限定域和协议的URL（例如 "http://www.example.com/blog/" ）。如果 link 未返回域，则整合框架将根据您的 SITE_ID setting 。

Atom Feed需要定义Feed的当前位置的 &lt； link rel =“self”&gt； 。

串联发布Atom和RSS Feed

有些开发人员喜欢提供Atom 和 RSS版本的Feed。使用Django很容易：只需创建 Feed 类的子类，并将 feed_type 设置为不同的类型即可。然后更新您的URLconf以添加额外的版本。

这里有一个完整的例子：

```
from django.contrib.syndication.views import Feed
from policebeat.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Police beat site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to police be
at central."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

注意

在此示例中，RSS提要使用 `description` ，而Atom提要使用 `subtitle` 。这是因为Atom Feed不提供Feed级"说明"，但他们提供"字幕"。

如果您在 `Feed` 类中提供 `description` ，Django将不会自动将其放入 `subtitle` 字幕和描述不一定是同一件事。而应定义 `subtitle` 属性。

在上面的示例中，我们只需将Atom Feed的 `subtitle` 设置为RSS Feed的 `description` ，因为它已经很短。

和附带的URLconf：

```
from django.conf.urls import url
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

urlpatterns = [
    # ...
    url(r'^sitenews/rss/$', RssSiteNewsFeed()),
    url(r'^sitenews/atom/$', AtomSiteNewsFeed()),
    # ...
]
```

Feed类引用

class `views.``Feed`

此示例说明 `Feed` 类的所有可能的属性和方法：

```
from django.contrib.syndication.views import Feed
from django.utils import feedgenerator
```

```
class ExampleFeed(Feed):

    # FEED TYPE -- Optional. This should be a class that subclas
ses
    # django.utils.feedgenerator.SyndicationFeed. This designate
s
    # which type of feed this should be: RSS 2.0, Atom 1.0, etc.
 If
    # you don't specify feed_type, your feed will be RSS 2.0\. T
his
    # should be a class, not an instance of the class.

    feed_type = feedgenerator.Rss201rev2Feed

    # TEMPLATE NAMES -- Optional. These should be strings
    # representing names of Django templates that the system sho
uld
    # use in rendering the title and description of your feed it
ems.
    # Both are optional. If a template is not specified, the
    # item_title() or item_description() methods are used instea
d.

    title_template = None
    description_template = None

    # TITLE -- One of the following three is required. The frame
work
    # looks for them in this order.

    def title(self, obj):
        """
 Takes the object returned by get_object() and returns the
 feed's title as a normal Python string.
 """

    def title(self):
        """
 Returns the feed's title as a normal Python string.
 """

    title = 'foo' # Hard-coded title.

    # LINK -- One of the following three is required. The framew
ork
    # looks for them in this order.

    def link(self, obj):
        """
 # Takes the object returned by get_object() and returns the URL
 # of the HTML version of the feed as a normal Python string.
 """
```

```
    def link(self):
        """
 Returns the URL of the HTML version of the feed as a normal Pyt
hon
 string.
 """

    link = '/blog/' # Hard-coded URL.

    # FEED_URL -- One of the following three is optional. The fr
amework
    # looks for them in this order.

    def feed_url(self, obj):
        """
 # Takes the object returned by get_object() and returns the fee
d's
 # own URL as a normal Python string.
 """

    def feed_url(self):
        """
 Returns the feed's own URL as a normal Python string.
 """

    feed_url = '/blog/rss/' # Hard-coded URL.

    # GUID -- One of the following three is optional. The framew
ork looks
    # for them in this order. This property is only used for Ato
m feeds
    # (where it is the feed-level ID element). If not provided,
the feed
    # link is used as the ID.

    def feed_guid(self, obj):
        """
 Takes the object returned by get_object() and returns the globa
lly
 unique ID for the feed as a normal Python string.
 """

    def feed_guid(self):
        """
 Returns the feed's globally unique ID as a normal Python string
.
 """

    feed_guid = '/foo/bar/1234' # Hard-coded guid.

    # DESCRIPTION -- One of the following three is required. The
 framework
```

```
    # looks for them in this order.

    def description(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 description as a normal Python string.
 """

    def description(self):
        """
 Returns the feed's description as a normal Python string.
 """

    description = 'Foo bar baz.' # Hard-coded description.

    # AUTHOR NAME --One of the following three is optional. The
framework
    # looks for them in this order.

    def author_name(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 author's name as a normal Python string.
 """

    def author_name(self):
        """
 Returns the feed's author's name as a normal Python string.
 """

    author_name = 'Sally Smith' # Hard-coded author name.

    # AUTHOR EMAIL --One of the following three is optional. The
 framework
    # looks for them in this order.

    def author_email(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 author's email as a normal Python string.
 """

    def author_email(self):
        """
 Returns the feed's author's email as a normal Python string.
 """

    author_email = 'test@example.com' # Hard-coded author email.

    # AUTHOR LINK --One of the following three is optional. The
```

```
framework
    # looks for them in this order. In each case, the URL should
 include
    # the "http://" and domain name.

    def author_link(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 author's URL as a normal Python string.
 """

    def author_link(self):
        """
 Returns the feed's author's URL as a normal Python string.
 """

    author_link = 'http://www.example.com/' # Hard-coded author
URL.

    # CATEGORIES -- One of the following three is optional. The
framework
    # looks for them in this order. In each case, the method/att
ribute
    # should return an iterable object that returns strings.

    def categories(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 categories as iterable over strings.
 """

    def categories(self):
        """
 Returns the feed's categories as iterable over strings.
 """

    categories = ("python", "django") # Hard-coded list of categ
ories.

    # COPYRIGHT NOTICE -- One of the following three is optional
. The
    # framework looks for them in this order.

    def feed_copyright(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 copyright notice as a normal Python string.
 """

    def feed_copyright(self):
```

```
        """
 Returns the feed's copyright notice as a normal Python string.
 """

    feed_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-co
ded copyright notice.

    # TTL -- One of the following three is optional. The framewo
rk looks
    # for them in this order. Ignored for Atom feeds.

    def ttl(self, obj):
        """
 Takes the object returned by get_object() and returns the feed'
s
 TTL (Time To Live) as a normal Python string.
 """

    def ttl(self):
        """
 Returns the feed's TTL as a normal Python string.
 """

    ttl = 600 # Hard-coded Time To Live.

    # ITEMS -- One of the following three is required. The frame
work looks
    # for them in this order.

    def items(self, obj):
        """
 Takes the object returned by get_object() and returns a list of
 items to publish in this feed.
 """

    def items(self):
        """
 Returns a list of items to publish in this feed.
 """

    items = ('Item 1', 'Item 2') # Hard-coded items.

    # GET_OBJECT -- This is required for feeds that publish diff
erent data
    # for different URL parameters. (See "A complex example" abo
ve.)

    def get_object(self, request, *args, **kwargs):
        """
 Takes the current request and the arguments from the URL, and
 returns an object represented by this feed. Raises
 django.core.exceptions.ObjectDoesNotExist on error.
 """
```

```
    # ITEM TITLE AND DESCRIPTION -- If title_template or
    # description_template are not defined, these are used inste
ad. Both are
    # optional, by default they will use the unicode representat
ion of the
    # item.

    def item_title(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 title as a normal Python string.
 """

    def item_title(self):
        """
 Returns the title for every item in the feed.
 """

    item_title = 'Breaking News: Nothing Happening' # Hard-coded
 title.

    def item_description(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 description as a normal Python string.
 """

    def item_description(self):
        """
 Returns the description for every item in the feed.
 """

    item_description = 'A description of the item.' # Hard-coded
 description.

    def get_context_data(self, **kwargs):
        """
 Returns a dictionary to use as extra context if either
 description_template or item_template are used.

 Default implementation preserves the old behavior
 of using {'obj': item, 'site': current_site} as the context.
 """

    # ITEM LINK -- One of these three is required. The framework
 looks for
    # them in this order.

    # First, the framework tries the two methods below, in
    # order. Failing that, it falls back to the get_absolute_url
()
    # method on each item returned by items().
```

```
    def item_link(self, item):
        """
 Takes an item, as returned by items(), and returns the item's U
RL.
 """

    def item_link(self):
        """
 Returns the URL for every item in the feed.
 """

    # ITEM_GUID -- The following method is optional. If not prov
ided, the
    # item's link is used by default.

    def item_guid(self, obj):
        """
 Takes an item, as return by items(), and returns the item's ID.
 """

    # ITEM_GUID_IS_PERMALINK -- The following method is optional
. If
    # provided, it sets the 'isPermaLink' attribute of an item's
    # GUID element. This method is used only when 'item_guid' is
    # specified.

    def item_guid_is_permalink(self, obj):
        """
 Takes an item, as returned by items(), and returns a boolean.
 """

    item_guid_is_permalink = False  # Hard coded value

    # ITEM AUTHOR NAME -- One of the following three is optional
. The
    # framework looks for them in this order.

    def item_author_name(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 author's name as a normal Python string.
 """

    def item_author_name(self):
        """
 Returns the author name for every item in the feed.
 """

    item_author_name = 'Sally Smith' # Hard-coded author name.

    # ITEM AUTHOR EMAIL --One of the following three is optional
. The
```

```
    # framework looks for them in this order.
    #
    # If you specify this, you must specify item_author_name.

    def item_author_email(self, obj):
        """
 Takes an item, as returned by items(), and returns the item's
 author's email as a normal Python string.
 """

    def item_author_email(self):
        """
 Returns the author email for every item in the feed.
 """

    item_author_email = 'test@example.com' # Hard-coded author e
mail.

    # ITEM AUTHOR LINK -- One of the following three is optional
. The
    # framework looks for them in this order. In each case, the
URL should
    # include the "http://" and domain name.
    #
    # If you specify this, you must specify item_author_name.

    def item_author_link(self, obj):
        """
 Takes an item, as returned by items(), and returns the item's
 author's URL as a normal Python string.
 """

    def item_author_link(self):
        """
 Returns the author URL for every item in the feed.
 """

    item_author_link = 'http://www.example.com/' # Hard-coded au
thor URL.

    # ITEM ENCLOSURE URL -- One of these three is required if yo
u're
    # publishing enclosures. The framework looks for them in thi
s order.

    def item_enclosure_url(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 enclosure URL.
 """

    def item_enclosure_url(self):
        """
```

```
Returns the enclosure URL for every item in the feed.
"""

    item_enclosure_url = "/foo/bar.mp3" # Hard-coded enclosure l
ink.

    # ITEM ENCLOSURE LENGTH -- One of these three is required if
 you're
    # publishing enclosures. The framework looks for them in thi
s order.
    # In each case, the returned value should be either an integ
er, or a
    # string representation of the integer, in bytes.

    def item_enclosure_length(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 enclosure length.
 """

    def item_enclosure_length(self):
        """
 Returns the enclosure length for every item in the feed.
 """

    item_enclosure_length = 32000 # Hard-coded enclosure length.

    # ITEM ENCLOSURE MIME TYPE -- One of these three is required
 if you're
    # publishing enclosures. The framework looks for them in thi
s order.

    def item_enclosure_mime_type(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 enclosure MIME type.
 """

    def item_enclosure_mime_type(self):
        """
 Returns the enclosure MIME type for every item in the feed.
 """

    item_enclosure_mime_type = "audio/mpeg" # Hard-coded enclosu
re MIME type.

    # ITEM PUBDATE -- It's optional to use one of these three. T
his is a
    # hook that specifies how to get the pubdate for a given ite
m.
    # In each case, the method/attribute should return a Python
    # datetime.datetime object.
```

```
    def item_pubdate(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 pubdate.
 """

    def item_pubdate(self):
        """
 Returns the pubdate for every item in the feed.
 """

    item_pubdate = datetime.datetime(2005, 5, 3) # Hard-coded pu
bdate.

    # ITEM UPDATED -- It's optional to use one of these three. T
his is a
    # hook that specifies how to get the updateddate for a given
 item.
    # In each case, the method/attribute should return a Python
    # datetime.datetime object.

    def item_updateddate(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 updateddate.
 """

    def item_updateddate(self):
        """
 Returns the updateddated for every item in the feed.
 """

    item_updateddate = datetime.datetime(2005, 5, 3) # Hard-code
d updateddate.

    # ITEM CATEGORIES -- It's optional to use one of these three
. This is
    # a hook that specifies how to get the list of categories fo
r a given
    # item. In each case, the method/attribute should return an
iterable
    # object that returns strings.

    def item_categories(self, item):
        """
 Takes an item, as returned by items(), and returns the item's
 categories.
 """

    def item_categories(self):
        """
 Returns the categories for every item in the feed.
 """
```

```
    item_categories = ("python", "django") # Hard-coded categori
es.

    # ITEM COPYRIGHT NOTICE (only applicable to Atom feeds) -- O
ne of the
    # following three is optional. The framework looks for them
in this
    # order.

    def item_copyright(self, obj):
        """
 Takes an item, as returned by items(), and returns the item's
 copyright notice as a normal Python string.
 """

    def item_copyright(self):
        """
 Returns the copyright notice for every item in the feed.
 """

    item_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-co
ded copyright notice.
```

低级框架

在后台，高级RSS框架使用较低级别的框架来生成订阅源的XML。此框架存在于单个模块中：django / utils / feedgenerator.py。

您自己使用此框架，用于生成较低级别的Feed。您还可以创建自定义Feed生成器子类，以与 `feed_type` `Feed` 选项一起使用。

联合供稿 classes

`feedgenerator` 模块包含基类：

- django.utils.feedgenerator.SyndicationFeed

和几个子类：

- django.utils.feedgenerator.RssUserland091Feed
- django.utils.feedgenerator.Rss201rev2Feed
- django.utils.feedgenerator.Atom1Feed

这三个类中的每一个都知道如何将某种类型的feed呈现为XML。他们共享这个接口：

`SyndicationFeed.__init__()`

使用给定的元数据字典初始化Feed，该元数据字典应用于整个Feed。必需的关键字参数为：

- `title`
- `link`
- `description`

还有一堆其他可选关键字：

- `language`
- `author_email`
- `author_name`
- `author_link`
- `subtitle`
- `categories`
- `feed_url`
- `feed_copyright`
- `feed_guid`
- `ttl`

您传递给 `__init__` 的任何其他关键字参数将存储在 `self.feed` 中以与自定义Feed生成器配合使用。

所有参数都应该是Unicode对象，除了 `categories` ，它应该是Unicode对象序列。

`SyndicationFeed.add_item()`

向具有给定参数的Feed中添加项目。

必需的关键字参数为：

- `title`
- `link`
- `description`

可选的关键字参数为：

- `author_email`
- `author_name`
- `author_link`
- `pubdate`
- `comments`
- `unique_id`
- `enclosure`
- `categories`
- `item_copyright`
- `ttl`
- `updateddate`

将为自定义Feed生成器存储额外的关键字参数。

所有参数，如果给定，应该是Unicode对象，除了：

- `pubdate` 应为Python `datetime` 对象。
- `updateddate` 应为Python `datetime` 对象。
- `enclosure` 应为 `django.utils.feedgenerator.Enclosure` 的实例。
- `categories` 应为Unicode对象序列。

New in Django 1.7:

已添加可选的 `updateddate` 参数。

`SyndicationFeed.write()`

Outputs the feed in the given encoding to outfile, which is a file-like object.

`SyndicationFeed.writeString()`

Returns the feed as a string in the given encoding.

例如，要创建Atom 1.0订阅源并将其打印到标准输出：

```
>>> from django.utils import feedgenerator
>>> from datetime import datetime
>>> f = feedgenerator.Atom1Feed(
...     title="My Weblog",
...     link="http://www.example.com/",
...     description="In which I write about what I ate today.",
...     language="en",
...     author_name="Myself",
...     feed_url="http://example.com/atom.xml")
>>> f.add_item(title="Hot dog today",
...     link="http://www.example.com/entries/1/",
...     pubdate=datetime.now(),
...     description="<p>Today I had a Vienna Beef hot dog. It wa
s pink, plump and perfect.</p>")
>>> print(f.writeString('UTF-8'))
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
...
</feed>
```

自定义Feed生成器

如果您需要生成自定义Feed格式，您有几个选项。

如果Feed格式是完全自定义的，您需要将 `SyndicationFeed` 作为子类，并完全替换 `write()` 和 `writeString()` 方法。

但是，如果Feed格式是RSS或Atom分析（即GeoRSS，Apple的iTunes podcast格式等），则您有了更好的选择。这些类型的Feed通常会向底层格式添加额外的元素和/或属性，并且有一组方法可以 `SyndicationFeed` 调用以获取这些额外的属性。因此，您可以对相应的Feed生成器类（ `Atom1Feed` 或 `Rss201rev2Feed` ）进行子类化，并扩展这些回调。他们是：

`SyndicationFeed.root_attributes(self, )`

Return a `dict` of attributes to add to the root feed element (`feed` / `channel`).

`SyndicationFeed.add_root_elements(self, handler)`

Callback to add elements inside the root feed element (`feed` / `channel`). `handler` is an `XMLGenerator` from Python's built-in SAX library; you'll call methods on it to add to the XML document in process.

`SyndicationFeed.item_attributes(self, item)`

Return a `dict` of attributes to add to each item (`item` / `entry`) element. The argument, `item` , is a dictionary of all the data passed to `SyndicationFeed.add_item()` .

`SyndicationFeed.add_item_elements(self, handler, item)`

Callback to add elements to each item (`item` / `entry`) element. `handler` and `item` are as above.

警告

如果您覆盖任何这些方法，请务必调用超类方法，因为它们为每种Feed格式添加了必需的元素。

例如，您可能开始实现iTunes RSS Feed生成器，如：

```
class iTunesFeed(Rss201rev2Feed):
    def root_attributes(self):
        attrs = super(iTunesFeed, self).root_attributes()
        attrs['xmlns:itunes'] = 'http://www.itunes.com/dtds/podc
ast-1.0.dtd'
        return attrs

    def add_root_elements(self, handler):
        super(iTunesFeed, self).add_root_elements(handler)
        handler.addQuickElement('itunes:explicit', 'clean')
```

显然，对于一个完整的自定义feed类，还有很多工作要做，但上面的例子应该展示基本的想法。

分页

Django提供了一些类来帮助你管理分页的数据 -- 也就是说，数据被分在不同页面中，并带有“上一页/下一页”标签。这些类位于`django/core/paginator.py`中。

示例

向`Paginator`提供对象的列表，以及你想为每一页分配的元素数量，它就会为你提供访问每一页上对象的方法：

```
>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)

>>> p.count
4
>>> p.num_pages
2
>>> p.page_range
[1, 2]

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
Traceback (most recent call last):
...
EmptyPage: That page contains no results
>>> page2.previous_page_number()
1
>>> page2.start_index() # The 1-based index of the first item on
 this page
3
>>> page2.end_index() # The 1-based index of the last item on th
is page
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

注意

注意你可以向 `Paginator` 提供一个列表或元组，Django的 `QuerySet` ，或者任何带有 `count()` 或 `__len__()` 方法的对象。当计算传入的对象所含对象的数量时， `Paginator` 会首先尝试调用 `count()` ，接着如果传入的对象没有 `count()` 方法则回退调用 `len()` 。这样会使类似于Django的 `QuerySet` 的对象使用更加高效的 `count()` 方法，如果存在的话。

使用 `Paginator`

这里有一些复杂一点的例子，它们在视图中使用 `Paginator` 来为查询集分页。我们提供视图以及相关的模板来展示如何展示这些结果。这个例子假设你拥有一个已经导入的 `Contacts` 模型。

视图函数看起来像是这样：

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25) # Show 25 contacts per page

    page = request.GET.get('page')
    try:
        contacts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer, deliver first page.
        contacts = paginator.page(1)
    except EmptyPage:
        # If page is out of range (e.g. 9999), deliver last page of results.
        contacts = paginator.page(paginator.num_pages)

    return render_to_response('list.html', {"contacts": contacts})
```

在 `list.html` 模板中，你会想要包含页面之间的导航，以及来自对象本身的任何有趣的信息：

```
<div class="pagination">
    <span class="step-links">

        <span class="current">
            Page  of .
        </span>

    </span>
</div>
```

Paginator objects

`Paginator` 类拥有以下构造器：

class `Paginator`(*object_list*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)[source]

所需参数

`object_list`

A list, tuple, Django `QuerySet`, or other sliceable object with a `count()` or `__len__()` method.

`per_page`

The maximum number of items to include on a page, not including orphans (see the `orphans` optional argument below).

可选参数

`orphans`

The minimum number of items allowed on the last page, defaults to zero. Use this when you don't want to have a last page with very few items. If the last page would normally have a number of items less than or equal to `orphans`, then those items will be added to the previous page (which becomes the last page) instead of leaving the items on a page by themselves. For example, with 23 items, `per_page=10`, and `orphans=3`, there will be two pages; the first page with 10 items and the second (and last) page with 13 items.

`allow_empty_first_page`

Whether or not the first page is allowed to be empty. If `False` and `object_list` is empty, then an `EmptyPage` error will be raised.

方法

`Paginator.``page` (*number*)[source]

返回在提供的下标处的 `Page` 对象，下标以1开始。如果提供的页码不存在，抛出 `InvalidPage` 异常。

属性

`Paginator.``count`

所有页面的对象总数。

注意

当计算 `object_list` 所含对象的数量时， `Paginator` 会首先尝试调用 `object_list.count()` 。如果 `object_list` 没有 `count()` 方法， `Paginator` 接着会回退使用 `len(object_list)` 。这样会使类似于Django's `QuerySet` 的对象使用更加便捷的 `count()` 方法，如果存在的话。

`Paginator.``num_pages`

页面总数。

`Paginator.``page_range`

页码的范围，从1开始，例如 `[1, 2, 3, 4]` 。

InvalidPage exceptions

exception `InvalidPage` [source]

异常的基类，当paginator传入一个无效的页码时抛出。

`Paginator.page()` 放回在所请求的页面无效（比如不是一个整数）时，或者不包含任何对象时抛出异常。通常，捕获 `InvalidPage` 异常就够了，但是如果你想更加精细一些，可以捕获以下两个异常之一：

exception `PageNotAnInteger` [source]

当向 `page()` 提供一个不是整数的值时抛出。

exception `EmptyPage` [source]

当向 `page()` 提供一个有效值，但是那个页面上没有任何对象时抛出。

这两个异常都是 `InvalidPage` 的子类，所以你可以通过简单的 `except InvalidPage` 来处理它们。

Page objects

你通常不需要手动构建 `Page` 对象 -- 你可以从 `Paginator.page()` 来获得它们。

class `Page` (*object_list*, *number*, *paginator*)[source]

当调用 `len()` 或者直接迭代一个页面的时候，它的行为类似于 `Page.object_list` 的序列。

方法

`Page.``has_next` ()[source]

Returns `True` if there's a next page.

`Page.``has_previous` ()[source]

如果有上一页，返回 `True` 。

`Page.``has_other_pages` ()[source]

如果有上一页或下一页，返回 `True` 。

`Page.``next_page_number` ()[source]

返回下一页的页码。如果下一页不存在，抛出 `InvalidPage` 异常。

`Page.``previous_page_number` ()[source]

返回上一页的页码。如果上一页不存在，抛出 `InvalidPage` 异常。

`Page.``start_index` ()[source]

返回当前页上的第一个对象，相对于分页列表的所有对象的序号，从1开始。比如，将五个对象的列表分为每页两个对象，第二页的 `start_index()` 会返回 `3` 。

`Page.``end_index` ()[source]

返回当前页上的最后一个对象，相对于分页列表的所有对象的序号，从1开始。 比如，将五个对象的列表分为每页两个对象，第二页的 `end_index()` 会返回 `4` 。

属性

`Page.``object_list`

当前页上所有对象的列表。

`Page.``number`

当前页的序号，从1开始。

`Page.``paginator`

相关的 `Paginator` 对象。

> 译者：Django 文档协作翻译小组，原文：Pagination。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

消息框架

在网页应用中，你经常需要在处理完表单或其它类型的用户输入后，显示一个通知消息（也叫做"flash message"）给用户。

对于这个功能，Django 提供基于Cookie 和会话的消息，无论是匿名用户还是认证的用户。其消息框架允许你临时将消息存储在请求中，并在接下来的请求（通常就是下一个请求）中提取它们并显示。每个消息都带有一个特定 `level` 标签，表示其优先级（例如 `info` 、 `warning` 或 `error` ）。

启用消息框架

消息框架的实现通过一个中间件 类和对应的*context processor*。

`django-admin startproject` 创建的默认 `settings.py` 已经包含启用消息框架功能需要的所有的设置：

- `INSTALLED_APPS` 中的 `'django.contrib.messages'` 。
- `MIDDLEWARE_CLASSES` 中的 `'django.contrib.sessions.middleware.SessionMiddleware'` 和 `'django.contrib.messages.middleware.MessageMiddleware'` 。

  默认的后端存储 依赖*sessions*。所以 `MIDDLEWARE_CLASSES` 中必须启用 `SessionMiddleware` 并出现在 `MessageMiddleware` 之前。
- `TEMPLATES` 设置中定义的 `DjangoTemplates` 的 `'context_processors'` 选项包含 `'django.contrib.messages.context_processors.messages'` 。

如果你不想使用消息框架，你可以删除 `INSTALLED_APPS` 中的 `'django.contrib.messages'` 、 `MIDDLEWARE_CLASSES` 中的 `MessageMiddleware` 和 `TEMPLATES` 中的 `messages` context processor。

配置消息框架引擎

后台存储

消息框架可以使用不同的后台存储临时消息。

Django 在 `django.contrib.messages` 中提供三个内建的存储类：

class `storage.session.``SessionStorage`

这个类存储所有的消息于请求的会话中。因此，它要求启用Django的 `contrib.sessions` 应用。

class `storage.cookie.``CookieStorage`

这个类存储消息数据于与Cookie 中（已经用一个安全的哈希进行签名以防止篡改）以在请求之间传递消息。如果Cookie 数据的大小将超过2048 字节，将丢弃旧的消息。

class `storage.fallback.``FallbackStorage`

这个类首先使用 `CookieStorage`，如果消息塞不进一个Cookie 中则使用 `SessionStorage` 。它同样要求启用Django 的 `contrib.sessions` 应用。

这个行为避免每次都写会话。在通常情况下，它提供的性能应该是最好的。

`FallbackStorage` 是默认的存储类。如果它不适合你的需要，你可以通过设置 `MESSAGE_STORAGE` 为它的完整导入路径选择另外一个存储类，例如：

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.cookie.Cookie
Storage'
```

class `storage.base.``BaseStorage`

如果想编写你自己的存储类，子类化 `django.contrib.messages.storage.base` 中的 `BaseStorage` 类并实现 `_get` 和 `_store` 方法。

消息级别

消息框架的级别是可配置的，与Python logging 模块类似。消息的级别可以让你根据类型进行分组，这样它们能够在不同的视图和模板中过滤或显示出来。

可以直接从 `django.contrib.messages` 导入的内建级别有：

Constant	Purpose
`DEBUG`	Development-related messages that will be ignored (or removed) in a production deployment
`INFO`	Informational messages for the user
`SUCCESS`	An action was successful, e.g. “Your profile was updated successfully”
`WARNING`	A failure did not occur but may be imminent
`ERROR`	An action was **not** successful or some other failure occurred

`MESSAGE_LEVEL` 设置可以用来改变记录的最小级别（它还可以在每个请求中修改）。小于这个级别的消息将被忽略。

消息的标签

消息的标签是一个字符串，表示消息的级别以及在视图中添加的其它标签（参见下文添加额外的消息标签）。标签存储在字符串中并通过空格分隔。通常情况下，消息的标签用于作为CSS 类来根据消息的类型定制消息的风格。默认情况下，每个级别具有一个标签，为其级别的字符串常量的小写：

Level Constant	Tag
`DEBUG`	`debug`
`INFO`	`info`
`SUCCESS`	`success`
`WARNING`	`warning`
`ERROR`	`error`

若要修改消息级别的默认标签，设置 `MESSAGE_TAGS` 为包含你想要修改的级别的字典。由于这扩展了默认标记，您只需要为要覆盖的级别提供标记：

```
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.INFO: '',
    50: 'critical',
}
```

在视图和模板中使用消息

`add_message` (*request*, *level*, *message*, *extra_tags=''*, *fail_silently=False*)

新增一条消息

新增一条消息，调用：

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

有几个快捷方法提供标准的方式来新增消息并带有常见的标签（这些标签通常表示消息的HTML 类型）：

```
messages.debug(request, '%s SQL statements were executed.' % cou
nt)
messages.info(request, 'Three credits remain in your account.')
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

显示消息

`get_messages` (*request*)

在你的模板中，像下面这样使用：

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif
%}>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

如果你正在使用context processor，你的模板应该通过 `RequestContext` 渲染。否则，需要确保 `messages` 在模板的Context 中可以访问。

即使你知道只有一条消息，你也应该仍然迭代 `messages` 序列，否则下个请求中的消息不会被清除。

New in Django 1.7.

Context processor 还提供一个 `DEFAULT_MESSAGE_LEVELS` 变量，它映射消息级别的名称到它们的数值：

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif
%}>
        {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}Im
portant: {% endif %}
        {{ message }}
    </li>
    {% endfor %}
</ul>
{% endif %}
```

在模板的外面，你可以使用 `get_messages()` ：

```
from django.contrib.messages import get_messages

storage = get_messages(request)
for message in storage:
    do_something_with_the_message(message)
```

例如，你可以获取所有的消息并在*JSONResponseMixin* 而不是 `TemplateResponseMixin` 中返回它们。

`get_messages()` 将返回配置的存储后台的一个实例。

的 `Message`

class `storage.base.``Message`

当浏览某个模板的消息列表时，你得到的其实是 `Message` 类的实例。它只是一个非常简单、只带很少属性的对象：

- `message`：消息的实际内容文本。
- `level`：一个整数，它描述了消息的类型 (请参阅上面 message levels一节).
- `tags`：一个字符串，它由该消息的所有标签 (`extra_tags` 和 `level_tag`) 组合而成，组合时用空格分割开这些标签。
- `extra_tags`：一个字符串，它由该消息的定制标签组合而成，并用空格分割。缺省为空。

New in Django 1.7.

- `level_tag`：代表该消息级别的字符串。该属性缺省由小写的关联常数名组成， 但当设置 `MESSAGE_TAGS` 参数时，可改变该规则。

创建自定义消息级别

消息级别只是整数，因此您可以定义自己的级别常量，并使用它们创建更多自定义的用户反馈，例如：

```
CRITICAL = 50

def my_view(request):
    messages.add_message(request, CRITICAL, 'A serious error occ
urred.')
```

在创建自定义消息级别时，应小心避免重载现有级别。内置级别的值为：

Level Constant	Value
DEBUG	10
INFO	20
SUCCESS	25
WARNING	30
ERROR	40

如果您需要识别HTML或CSS中的自定义级别，则需要通过 MESSAGE_TAGS 设置提供映射。

注意

如果要创建可重复使用的应用程序，建议仅使用内置的消息级别，而不依赖于任何自定义级别。

在每个请求中修改最小的记录级别

每个请求都可以通过 set_level 方法设置最小记录级别:

```
from django.contrib import messages

# Change the messages level to ensure the debug message is added
.
messages.set_level(request, messages.DEBUG)
messages.debug(request, 'Test message...')

# In another request, record only messages with a level of WARNI
NG and higher
messages.set_level(request, messages.WARNING)
messages.success(request, 'Your profile was updated.') # ignored
messages.warning(request, 'Your account is about to expire.') #
recorded

# Set the messages level back to default.
messages.set_level(request, None)
```

与此相似，当前有效的记录级别可以用 get_level 方法获取:

```
from django.contrib import messages
current_level = messages.get_level(request)
```

有关最小记录级别相关的函数信息，请参阅上面 Message levels 一节.

添加额外的消息标签

要更直接地控制消息标签，您可以选择为任何添加方法提供包含额外标签的字符串：

```
messages.add_message(request, messages.INFO, 'Over 9000!',
                     extra_tags='dragonball')
messages.error(request, 'Email box full', extra_tags='email')
```

在该级别的默认标记之前添加额外的标记，并以空格分隔。

当消息框架被禁止时，失败静悄悄

如果您撰写的是可重复使用的应用程式（或其他碎片代码），但想要包含讯息功能，但又不想要您的使用者启用（如果他们不想使用），您可以传送额外的关键字参数 `fail_silently = True` 指向任何 `add_message` 方法系列。举个例子

```
messages.add_message(request, messages.SUCCESS, 'Profile details
 updated.',
                     fail_silently=True)
messages.info(request, 'Hello world.', fail_silently=True)
```

注意

设置 `fail_silently=True` 只会隐藏消息框架禁用时会出现的 `MessageFailure` ，并尝试使用 `add_message` 。它不会隐藏可能由于其他原因发生的故障。

在基于类的视图中添加消息

class `views.``SuccessMessageMixin`

向基于 `FormView` 的类添加一条成功的消息

`get_success_message` (*cleaned_data*)

`cleaned_data` 是表单中的清洁数据，用于字符串格式化

示例 **views.py**：

```
from django.contrib.messages.views import SuccessMessageMixin
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(SuccessMessageMixin, CreateView):
    model = Author
    success_url = '/success/'
    success_message = "%(name)s was created successfully"
```

字符串插值可以使用 `%(field_name)s` 语法访问 `form` 中的清洁数据。对于 ModelForms，如果你需要访问保存的 `object` 中的字段，可以覆盖 `get_success_message()` 方法。

ModelForms 的示例views.py：

```
from django.contrib.messages.views import SuccessMessageMixin
from django.views.generic.edit import CreateView
from myapp.models import ComplicatedModel

class ComplicatedCreate(SuccessMessageMixin, CreateView):
    model = ComplicatedModel
    success_url = '/success/'
    success_message = "%(calculated_field)s was created successf
ully"

    def get_success_message(self, cleaned_data):
        return self.success_message % dict(cleaned_data,
                                           calculated_field=self
.object.calculated_field)
```

消息过期

消息被标记为在存储实例被迭代时被清除（并且当响应被处理时被清除）。

为了避免消息被清除，您可以在迭代后将消息存储设置为 `False`：

```
storage = messages.get_messages(request)
for message in storage:
    do_something_with(message)
storage.used = False
```

并行请求的行为

由于Cookie（以及会话）的工作方式，使用**Cookie**或会话的任何后端的行为在同一客户端发出并行设置或获取消息的多个请求时未定义。例如，如果客户端在第一窗口重定向之前发起在一个窗口（或标签）中创建消息并且然后在另一个窗口中获取另一个单元消息的请求，则该消息可以出现在第二窗口中而不是第一窗口中窗口，它可能是预期的。

简而言之，当涉及来自相同客户端的多个同时请求时，不能保证消息被传递到创建它们的相同窗口，在某些情况下根本不传递。注意，这在大多数应用中通常不是问题，并且在HTML5中将成为非问题，其中每个窗口/选项卡将具有其自己的浏览上下文。

设置

几个*settings*可让您控制邮件行为：

- MESSAGE_LEVEL
- MESSAGE_STORAGE
- MESSAGE_TAGS

New in Django 1.7.

对于使用Cookie的后端，Cookie的设置取自会话Cookie设置：

- SESSION_COOKIE_DOMAIN
- SESSION_COOKIE_SECURE
- SESSION_COOKIE_HTTPONLY

序列化Django对象

通常情况下，这种形式是基于文本的，它被用来发送Django的数据，当然，序列化处理的形式也有例外（基于文本或者相反）。

参见

如果您只是想从表中获取一些数据到序列化形式，可以使用 `dumpdata` 管理命令。

序列化数据

从最高层面来看,序列化数据是一项非常简单的操作

```
from django.core import serializers
data = serializers.serialize("xml", SomeModel.objects.all())
```

传递给 `serialize` 方法的参数有二：一个序列化目标格式(参见 Serialization formats) ，另外一个是序列号的对象 `QuerySet` . (事实上，第二个参数可以是任何可迭代的Django Model实例，但它很多情况下就是一个QuerySet).

`django.core.serializers.``get_serializer` (*format*)

你也可以直接序列化一个对象

```
XMLSerializer = serializers.get_serializer("xml")
xml_serializer = XMLSerializer()
xml_serializer.serialize(queryset)
data = xml_serializer.getvalue()
```

如果您想将数据直接序列化为类似文件的对象（其中包含 `HttpResponse` ），则此选项非常有用：

```
with open("file.xml", "w") as out:
    xml_serializer.serialize(SomeModel.objects.all(), stream=out
)
```

注意

调用具有未知格式的 `get_serializer()` 会产生 `django.core.serializers.SerializerDoesNotExist` 异常。

字段子集

如果只想要序列化一部分字段，可以为序列化程序指定 `字段` 参数：

```
from django.core import serializers
data = serializers.serialize('xml', SomeModel.objects.all(), fie
lds=('name','size'))
```

在本示例中，只有每个模型的 `名称` 和 `尺寸` 属性都将被序列化。

注意

根据你定义的模型，您会发现不能反序列化一个只序列化其字段子集的模型。如果序列化对象未指定模型所必需的所有字段，则解序器将无法保存反序列化的实例。

Inherited Models 继承模型

如果你定义的模型继承了*abstract base class*, 你不需要对它做任何特别的处理。只需对需要序列化的对象调用serializers，就能输出该对象完整的字段。

但是，如果您有使用*multi-table inheritance*的模型，则还需要序列化模型的所有基类。这是因为只有在模型上本地定义的字段才会被序列化。例如，考虑以下模型：

```
class Place(models.Model):
    name = models.CharField(max_length=50)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
```

如果你只序列化餐厅模型：

```
data = serializers.serialize('xml', Restaurant.objects.all())
```

序列化输出上的字段将只包含 `serves_hot_dogs` 属性。基类的 `name` 属性将被忽略。

为了完全串行化您的 `Restaurant` 实例，您还需要将 `Place` 模型序列化：

```
all_objects = list(Restaurant.objects.all()) + list(Place.object
s.all())
data = serializers.serialize('xml', all_objects)
```

反序列化数据

反序列化数据也是一个相当简单的操作：

```
for obj in serializers.deserialize("xml", data):
    do_something_with(obj)
```

如你所见，`deserialize` 函数采用与 `serialize` 相同的格式参数，一个字符串或数据流，并返回一个迭代器。

然而，在这里它有点复杂。`deserialize` 迭代器返回的对象不是简单的Django对象。相反，它们是包装已创建但未保存的对象和任何关联关系数据的特殊 `DeserializedObject` 实例。

调用 `DeserializedObject.save()` 将对象保存到数据库。

注意

如果序列化数据中的 `pk` 属性不存在或为null，则新实例将保存到数据库。

这确保了反序列化是一种非破坏性操作，即使序列化表示中的数据与数据库中当前的数据不匹配。通常，使用这些 `DeserializedObject` 实例看起来像：

```
for deserialized_object in serializers.deserialize("xml", data):
    if object_should_be_saved(deserialized_object):
        deserialized_object.save()
```

换句话说，通常的用法是检查反序列化的对象，以确保它们是“适当的”用于保存之前这样做。当然，如果你相信你的数据源，你可以保存对象，继续前进。

Django对象本身可以被检查为 `deserialized_object.object` 。如果模型中不存在序列化数据中的字段，则会出现 `DeserializationError` ，除非 `ignorenonexistent` 参数以 `True`

```
serializers.deserialize("xml", data, ignorenonexistent=True)
```

序列化格式

Django支持多种序列化格式，其中一些格式要求您安装第三方Python模块：

Identifier	Information
`xml`	Serializes to and from a simple XML dialect.
`json`	Serializes to and from JSON.
`yaml`	Serializes to YAML (YAML Ain’t a Markup Language). This serializer is only available if PyYAML is installed.

XML

基本的XML序列化格式很简单：

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="123" model="sessions.session">
        <field type="DateTimeField" name="expire_date">2013-01-1
6T08:16:59.844560+00:00</field>
        <!-- ... -->
    </object>
</django-objects>
```

序列化或反序列化的对象的整个集合由包含多个 `<object>` 元素的 `<django-objects>` -tag表示。每个这样的对象有两个属性："pk"和"model"，后者由应用程序的名称（"sessions"）和用点分隔的模型的小写名称（"会话"）表示。

对象的每个字段被序列化为 `<field>` - 运行字段"type"和"name"的元素。元素的文本内容表示应存储的值。

外键和其他关系字段的处理方式略有不同：

```
<object pk="27" model="auth.permission">
    <!-- ... -->
    <field to="contenttypes.contenttype" name="content_type" rel
="ManyToOneRel">9</field>
    <!-- ... -->
</object>
```

在这个例子中，我们指定了auth。PK 27的权限对象具有内容类型的外键。ContentType实例与PK 9。

ManyToMany关系导出为绑定它们的模型。例如，auth。用户模型与auth有这样的关系。权限模型：

```
<object pk="1" model="auth.user">
    <!-- ... -->
    <field to="auth.permission" name="user_permissions" rel="Man
yToManyRel">
        <object pk="46"></object>
        <object pk="47"></object>
    </field>
</object>
```

此示例将给定用户与具有PK 46和47的权限模型链接。

JSON

当保持与之前相同的示例数据时，将以下列方式序列化为JSON：

```
[
    {
        "pk": "4b678b301dfd8a4e0dad910de3ae245b",
        "model": "sessions.session",
        "fields": {
            "expire_date": "2013-01-16T08:16:59.844Z",
            ...
        }
    }
]
```

这里的格式比使用XML简单一点。整个集合仅表示为数组，对象由具有三个属性的JSON对象表示："pk"，"model"和"fields"。"fields"再次是一个对象，分别将每个字段的名称和值分别作为property和property-value。

外键只是将链接对象的PK作为属性值。ManyToMany关系被序列化为定义它们的模型，并被表示为PK列表。

日期和日期时间相关类型由JSON序列化程序以特殊方式处理，以使格式与ECMA-262兼容。

请注意，并非所有Django输出都可以未修改地传递给 `json` 。特别地，*lazy translation objects*需要为它们写入特殊编码器。这样的东西会工作：

```
from django.utils.functional import Promise
from django.utils.encoding import force_text
from django.core.serializers.json import DjangoJSONEncoder

class LazyEncoder(DjangoJSONEncoder):
    def default(self, obj):
        if isinstance(obj, Promise):
            return force_text(obj)
        return super(LazyEncoder, self).default(obj)
```

另请注意，GeoDjango提供了一个*customized GeoJSON serializer*。

YAML

YAML序列化看起来非常类似于JSON。对象列表被序列化为具有键"pk"，"model"和"fields"的序列映射。每个字段再次是一个映射，其中键是字段的名称和值的值：

```
-   fields: {expire_date: !!timestamp '2013-01-16 08:16:59.844560+00:00'}
    model: sessions.session
    pk: 4b678b301dfd8a4e0dad910de3ae245b
```

参考字段再次仅由PK或PK序列表示。

自然钥匙

外键和多对多关系的默认序列化策略是序列化关系中对象的主键的值。这个策略适用于大多数对象，但在某些情况下可能会导致困难。

考虑具有引用 `ContentType` 的外键的对象列表的情况。如果你要序列化一个引用内容类型的对象，那么你需要有一种方法来引用该内容类型。由于 `ContentType` 对象在数据库同步过程中由Django自动创建，所以给定内容类型的主键不容易预测；它将取决于执行 `migrate` 的方式和时间。这对于所有自动生成对象的模型都是如此，尤其包括 `Permission` ， `Group` 和 `User` 。

警告

您不应将自动生成的对象包括在夹具或其他序列化数据中。偶尔，夹具中的主键可能与数据库中的主键匹配，加载夹具将没有效果。在更可能的情况下，它们不匹配，夹具加载将失败，并出现 `IntegrityError` 。

还有方便的事情。整数id并不总是最方便的引用对象的方法；有时，更自然的参考将是有帮助的。

正是由于这些原因，Django提供了自然键。自然键是可用于唯一地标识对象实例而不使用主键值的值的元组。

自然键的反序列化

考虑以下两个模型：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        unique_together = (('first_name', 'last_name'),)

class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person)
```

通常，`Book`的序列化数据将使用整数来引用作者。例如，在JSON中，一本书可能被序列化为：

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {
        "name": "Mostly Harmless",
        "author": 42
    }
}
...
```

这不是一个特别自然的方式来引用作者。它要求你知道作者的主键值；它还要求这个主键值是稳定和可预测的。

然而，如果我们添加自然键处理到人，灯具变得更加人性化。要添加自然键处理，请使用`get_by_natural_key()`方法为人定义默认管理器。在Person的情况下，良好的自然键可能是名和姓：

```
from django.db import models

class PersonManager(models.Manager):
    def get_by_natural_key(self, first_name, last_name):
        return self.get(first_name=first_name, last_name=last_na
me)

class Person(models.Model):
    objects = PersonManager()

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        unique_together = (('first_name', 'last_name'),)
```

现在书可以使用该自然键来引用 `Person` 对象：

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {
        "name": "Mostly Harmless",
        "author": ["Douglas", "Adams"]
    }
}
...
```

当您尝试加载此序列化数据时，Django将使用 `get_by_natural_key()` 方法解析 `[“Douglas”， “Adams”]` 转换为实际 `Person` 对象的主键。

注意

无论用于自然键的任何字段都必须能够唯一地标识对象。这通常意味着您的模型将在自然键中的一个或多个字段中具有唯一性子句（单个字段上的unique = True，或多个字段中的 `unique_together` ）。但是，不需要在数据库级别强制实施唯一性。如果您确定一组字段有效地唯一，您仍然可以将这些字段用作自然键。

New in Django 1.7.

没有主键的对象的反序列化将始终检查模型的管理器是否具有 `get_by_natural_key()` 方法，如果是，则使用它来填充反序列化对象的主键。

自然键的序列化

那么在序列化对象时，如何让Django发射一个自然的键呢？首先，你需要添加另一个方法 - 这一次到模型本身：

```
class Person(models.Model):
    objects = PersonManager()

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    def natural_key(self):
        return (self.first_name, self.last_name)

    class Meta:
        unique_together = (('first_name', 'last_name'),)
```

该方法应该总是返回一个自然键元组 - 在这个例子中，（第一个 名称， 最后 t4> 。然后，当您调用 `serializers.serialize()` 时，您提供 `use_natural_foreign_keys=True` 或 `use_natural_primary_keys=True`

```
>>> serializers.serialize('json', [book1, book2], indent=2,
...         use_natural_foreign_keys=True, use_natural_primary_keys
=True)
```

当指定 `use_natural_foreign_keys=True` 时，Django将使用 `natural_key()` 方法将任何外键引用序列化为定义该方法的类型的对象。

当指定 `use_natural_primary_keys=True` 时，Django不会在此对象的序列化数据中提供主键，因为它可以在反序列化期间计算：

```
...
{
    "model": "store.person",
    "fields": {
        "first_name": "Douglas",
        "last_name": "Adams",
        "birth_date": "1952-03-11",
    }
}
...
```

当您需要将序列化数据加载到现有数据库中，并且不能保证序列化主键值尚未使用时，这是非常有用的，并且不需要确保反序列化对象保留相同的主键。

如果您使用 `dumpdata` 生成序列化数据，请使用 `--natural-foreign` 和 `--natural-primary` 命令行标志生成自然键。

注意

您不需要同时定义 `natural_key()` 和 `get_by_natural_key()` 。如果您不希望Django在序列化过程中输出自然键，但您希望保留加载自然键的能力，那么您可以选择不实现 `natural_key()` 方法。

相反，如果（由于某种奇怪的原因）你希望Django在序列化过程中输出自然键，但不能能够加载这些键值，只是不要定义 `get_by_natural_key()`

Changed in Django 1.7:

以前，对于 `serializers.serialize()` 和<cite>-n</cite>或 <cite>- 自然</cite>命令行只有 `use_natural_keys` 这些已被弃用，支持 `use_natural_foreign_keys` 和 `use_natural_primary_keys` 参数和相应的 `--natural-foreign` 和 `--natural-primary` 选项 `dumpdata` 。

原始参数和命令行标志保持向后兼容性，并映射到新的 `use_natural_foreign_keys` 参数和<cite>-natural-foreign</cite>命令行标志。他们将在Django 1.9中删除。

序列化期间的依赖关系

由于自然键依赖于数据库查找来解析引用，因此重要的是数据在引用之前存在。您不能使用自然键创建“前向引用” - 您引用的数据必须存在，然后才能包含该数据的自然键引用。

为了适应此限制，使用 `--natural-foreign` 选项的 `dumpdata` 调用将序列化之前使用 `natural_key()` 方法序列化任何模型标准主键对象。

然而，这可能并不总是足够。如果你的自然键引用另一个对象（通过使用外键或自然键对另一个对象作为自然键的一部分），那么你需要能够确保自然键所依赖的对象在序列化数据中出现在自然键之前需要它们。

要控制此顺序，可以在 `natural_key()` 方法上定义依赖关系。您可以通过在 `natural_key()` 方法本身设置 `dependencies` 属性来实现。

例如，让我们从上面的例子中添加一个自然键到 `Book` 模型：

```
class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person)

    def natural_key(self):
        return (self.name,) + self.author.natural_key()
```

`Book` 的自然键是其名称和作者的组合。这意味着 `Person` 必须在 `Book` 之前序列化。要定义这个依赖，我们添加一行：

```
def natural_key(self):
    return (self.name,) + self.author.natural_key()
natural_key.dependencies = ['example_app.person']
```

此定义确保所有 `Person` 对象在任何 `Book` 对象之前序列化。反过来，在 `Person` 和 `Book` 序列化之后，引用 `Book` 的任何对象将被序列化。

如何使用会话

Django 提供对匿名会话的完全支持。其会话框架让你根据各个站点的访问者存储和访问任意数据。它在服务器端存储数据并抽象Cookie 的发送和接收。Cookie 包含会话的ID —— 不是数据本身（除非你使用基于Cookie 的后端）。

启用会话

会话是通过一个中间件实现的。

为了启用会话功能，需要这样做：

编辑 `MIDDLEWARE_CLASSES` 设置并确保它包含' `django.contrib.sessions.middleware.SessionMiddleware` '。 `django-ad` 创建的默认的 `settings.py` 已经启用 `SessionMiddleware` 。如果你不想使用会话，你也可以从 `MIDDLEWARE_CLASSES` 中删除 `SessionMiddleware` 行，并从 `INSTALLED_APPS` 中删除' `django.contrib.sessions` '。它将节省一些性能消耗。

配置会话引擎

默认情况下，Django 存储会话到你的数据库中（使用 `django.contrib.sessions.models.Session` 模型）。虽然这很方便，但是在某些架构中存储会话在其它地方会更快，所以可以配置Django 来存储会话到你的文件系统上或缓存中。

使用数据库支持的会话

如果你想使用数据库支持的会话，你需要添加' `django.contrib.sessions` '到你的 `INSTALLED_APPS` 设置中。

在配置完成之后，请运行 `manage.py migrate` 来安装保存会话数据的一张数据库表。

使用基于缓存的会话

为了更好的性能，你可能想使用一个基于缓存的会话后端。

为了使用Django 的缓存系统来存储会话数据，你首先需要确保你已经配置好你的缓存；详细信息参见缓存的文档。

> 警告
>
> 你应该只在使用Memcached 缓存系统时才使用基于缓存的会话。基于本地内存的缓存系统不会长时间保留数据，所以不是一个好的选择，而且直接使用文件或数据库会话比通过文件或数据库缓存系统要快。另外，基于本地内存的缓存系统不是多进程安全的，所以对于生产环境可能不是一个好的选择。

如果你在 `CACHES` 中定义多个缓存，Django 将使用默认的缓存。若要使用另外一种缓存，请设置 `SESSION_CACHE_ALIAS` 为该缓存的名字。

配置好缓存之后，对于如何在缓存中存储数据你有两个选择：

- 对于简单的缓存会话存储，可以设置 `SESSION_ENGINE` 为" `django.contrib.sessions.backends.cache` " 。此时会话数据将直接存储在你的缓存中。然而，缓存数据将可能不会持久：如果缓存填满或者缓存服务器重启，缓存数据可能会被清理掉。
- 若要持久的缓存数据，可以设置 `SESSION_ENGINE` 为" `django.contrib.sessions.backends.cached_db` "。它的写操作使用缓存 —— 对缓存的每次写入都将再写入到数据库。对于读取的会话，如果数据不在缓存中，则从数据库读取。

两种会话的存储都非常快，但是简单的缓存更快，因为它放弃了持久性。大部分情况下， `cached_db` 后端已经足够快，但是如果你需要榨干最后一点的性能，并且接收让会话数据丢失，那么你可使用 `cache` 后端。

如果你使用 `cached_db` 会话后端，你还需要遵循使用数据库支持的会话中的配置说明。

```
Changed in Django 1.7:

在1.7 版之前，`cached_db` 永远使用`default`缓存而不是`SESSION_CACHE_A
LIAS`。
```

使用基于文件的缓存

要使用基于文件的缓存，请设置 `SESSION_ENGINE` 为" `django.contrib.sessions.backends.file` "。

你可能还想设置 `SESSION_FILE_PATH` （它的默认值来自 `tempfile.gettempdir()` 的输出，大部分情况是 `/tmp` ）来控制Django在哪里存储会话文件。请保证你的Web 服务器具有读取和写入这个位置的权限。

使用基于Cookie 的会话

要使用基于Cookie 的会话，请设置 `SESSION_ENGINE` 为" `django.contrib.sessions.backends.signed_cookies` "。此时，会话数据的存储将使用Django 的加密签名 工具和 `SECRET_KEY` 设置。

> 注
>
> 建议保留SESSION_COOKIE_HTTPONLY 设置为True 以防止从JavaScript 中访问存储的数据。
>
> 警告
>
> 如果 `SECRET_KEY` 没有保密并且你正在使用 `PickleSerializer` ，这可能导致远端执行任意的代码。
>
> 拥有 `SECRET_KEY` 的攻击者不仅可以生成篡改的会话数据而你的站点将会信任这些数据，而且可以远程执行任何代码，就像数据是通过pickle 序列化过的一样。
>
> 如果你使用基于Cookie 的会话，请格外注意你的安全秘钥对于任何可以远程访问的系统都是永远完全保密的。
>
> 会话数据经过签名但没有加密。
>
> 如果使用基于Cookie的会话，则会话数据可以被客户端读取。
>
> MAC（消息认证码）被用来保护数据不被客户端修改，所以被篡改的会话数据将是变成不合法的。如果保存Cookie的客户端（例如你的浏览器）不能保存所有的会话Cookie或丢失数据，会话同样会变得不合法。尽管Django 对数据进行压缩，仍然完全有可能超过每个Cookie 常见的4096 个字节的限制 。
>
> 没有更新保证
>
> 还要注意，虽然MAC可以保证数据的权威性（由你的站点生成，而不是任何其他人）和完整性（包含全部的数据并且是正确的），它不能保证是最新的，例如返回给你发送给客户端的最新的数据。这意味着对于某些会话数据的使用，基于Cookie 可能让你受到重放攻击。其它方式的会话后端在服务器端保存每个会话并在用户登出时使它无效，基于Cookie 的会话在用户登出时不会失效。因此，如果一个攻击者盗取用户的Cookie，它们可以使用这个Cookie 来以这个用户登录即使用户已登出。Cookies 只能被当做是“过期的”，如果它们比你的SESSION_COOKIE_AGE要旧。
>
> 性能
>
> 最后，Cookie 的大小对你的网站的速度 有影响。

在视图中使用会话

当 `SessionMiddleware` 激活时，每个 `HttpRequest` 对象 —— 传递给Django 视图函数的第一个参数 —— 将具有一个 `session` 属性，它是一个类字典对象。

你可以在你的视图中任何地方读取并写入 `request.session` 。你可以多次编辑它。

```
class backends.base.SessionBase
```

这是所有会话对象的基类。它具有以下标准的字典方法：

`__getitem__(key)`

例如：`fav_color = request.session['fav_color']`

`__setitem__(key, value)`

例如：`request.session['fav_color'] = 'blue'`

`__delitem__(key)`

例如：`del request.session['fav_color']`。如果给出的key 在会话中不存在，将抛出 `KeyError`。

`__contains__(key)`

例如：`'fav_color' in request.session`

`get(key, default=None)`

例如：`fav_color = request.session.get('fav_color', 'red')`

`pop(key)`

例如：`fav_color = request.session.pop('fav_color')`

`keys()`

`items()`

`setdefault()`

`clear()`

它还具有这些方法：

`flush()`

删除当前的会话数据并删除会话的Cookie。这用于确保前面的会话数据不可以再次被用户的浏览器访问（例如，`django.contrib.auth.logout()` 函数中就会调用它）。

```
Changed in Django 1.8:

删除会话Cookie 是Django 1.8 中的新行为。以前，该行为用于重新生成会话中的
值，这个值会在Cookie 中发回给用户。
```

`set_test_cookie()`

设置一个测试的Cookie 来验证用户的浏览器是否支持Cookie。因为Cookie 的工作方式，只有到用户的下一个页面才能验证。更多信息参见下文的设置测试的Cookie。

`test_cookie_worked()`

返回 `True` 或 `False` ，取决于用户的浏览器时候接受测试的Cookie。因为Cookie的工作方式，你必须在前面一个单独的页面请求中调用 `set_test_cookie()` 。更多信息参见下文的设置测试的Cookie。

`delete_test_cookie()`

删除测试的Cookie。使用这个函数来自己清理。

`set_expiry(value)`

设置会话的超时时间。你可以传递一系列不同的值：

- 如果 `value` 是一个整数，会话将在这么多秒没有活动后过期。例如，调用 `request.session.set_expiry(300)` 将使得会话在5分钟后过期。
- 若果value 是一个 `datetime` 或 `timedelta` 对象，会话将在这个指定的日期/时间过期。注意 `datetime` 和 `timedelta` 值只有在你使用P `ickleSerializer` 时才可序列化。
- 如果 `value` 为0，那么用户会话的Cookie将在用户的浏览器关闭时过期。
- 如果 `value` 为 `None` ，那么会话转向使用全局的会话过期策略。

过期的计算不考虑读取会话的操作。会话的过期从会话上次修改的时间开始计算。

`get_expiry_age()`

返回会话离过期的秒数。对于没有自定义过期的会话（或者设置为浏览器关闭时过期的会话），它将等于 `SESSION_COOKIE_AGE` 。

该函数接收两个可选的关键字参数：

- `modification` ：会话的最后一次修改时间，类型为一个 `datetime` 对象。默认为当前的时间。
- `expiry` ：会话的过期信息，类型为一个 `datetime` 对象、一个整数（以秒为单位）或 `None` 。默认为通过 `set_expiry()` 保存在会话中的值，如果没有则为 `None` 。

`get_expiry_date()`

返回过期的日期。对于没有自定义过期的会话（或者设置为浏览器关闭时过期的会话），它将等于从现在开始 `SESSION_COOKIE_AGE` 秒后的日期。

这个函数接受与 `get_expiry_age()` 一样的关键字参数。

`get_expire_at_browser_close()`

返回 `True` 或 `False` ，取决于用户的会话Cookie在用户浏览器关闭时会不会过期。

`clear_expired()`

从会话的存储中清除过期的会话。这个类方法被 `clearsessions` 调用。

`cycle_key()`

创建一个新的会话，同时保留当前的会话数据。 `django.contrib.auth.login()` 调用这个方法来减缓会话的固定。

会话的序列化

在1.6 版以前，在保存会话数据到后端之前Django 默认使用pickle 来序列化它们。如果你使用的是签名的Cookie 会话后端 并且 `SECRET_KEY` 被攻击者知道（Django 本身没有漏洞会导致它被泄漏），攻击者就可以在会话中插入一个字符串，在unpickle 之后可以在服务器上执行任何代码。在因特网上这个攻击技术很简单并很容易查到。尽管Cookie 会话的存储对Cookie 保存的数据进行了签名以防止篡改， `SECRET_KEY` 的泄漏会立即使得可以执行远端的代码。

这种攻击可以通过JSON而不是pickle序列化会话数据来减缓。为了帮助这个功能，Django 1.5.3 引入一个新的设置， `SESSION_SERIALIZER` ，来自定义会话序列化的格式。为了向后兼容，这个设置在Django 1.5.x 中默认为 `django.contrib.sessions.serializers.PickleSerializer` ，但是为了增强安全性，在Django 1.6 中默认为 `django.contrib.sessions.serializers.JSONSerializer` 。即使在编写你自己的序列化方法讲述的说明中，我们也强烈建议依然使用JSON 序列化，特别是在你使用的是Cookie 后端时。

绑定的序列化方法

`class serializers.JSONSerializer`

对 `django.core.signing` 中的JSON 序列化方法的一个包装。只可以序列基本的数据类型。

另外，因为JSON 只支持字符串作为键，注意使用非字符串作为 `request.session` 的键将不工作：

```
>>> # initial assignment
>>> request.session[0] = 'bar'
>>> # subsequent requests following serialization & deserializat
ion
>>> # of session data
>>> request.session[0]  # KeyError
>>> request.session['0']
'bar'
```

参见编写你自己的序列化器 一节以获得更多关于JSON 序列化的限制。

`class serializers.PickleSerializer`

支持任意Python 对象，但是正如上面描述的，可能导致远端执行代码的漏洞，如果攻击者知道了 `SECRET_KEY` 。

编写你自己的序列化器

注意，与 `PickleSerializer` 不同， `JSONSerializer` 不可以处理任意的Python 数据类型。这是常见的情况，需要在便利性和安全性之间权衡。如果你希望在JSON 格式的会话中存储更高级的数据类型比如 `datetime` 和 `Decimal` ，你需要编写一个自定义的序列化器（或者在保存它们到 `request.session` 中之前转换这些值到一个可JSON 序列化的对象）。虽然序列化这些值相当简单直接（ `django.core.serializers.json.DateTimeAwareJSONEncoder` 可能帮得上忙），编写一个解码器来可靠地取出相同的内容却能困难。例如，返回一个 `datetime` 时，它可能实际上是与 `datetime` 格式碰巧相同的一个字符串）。

你的序列化类必须实现两个方法， `dumps(self, obj)` 和 `loads(self, data)` 来分别序列化和去序列化会话数据的字典。

会话对象指南

`在request.session` 上使用普通的Python 字符串作为字典的键。这主要是为了方便而不是一条必须遵守的规则。 以一个下划线开始的会话字典的键被Django保留作为内部使用。 不要新的对象覆盖 `request.session` ，且不要访问或设置它的属性。要像Python 字典一样使用它。

例子

下面这个简单的视图在一个用户提交一个评论后设置 `has_commented` 变量为 `True` 。它不允许一个用户多次提交评论：

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

登录站点一个“成员”的最简单的视图：

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't m
atch.")
```

...下面是登出一个成员的视图，已经上面的login()：

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

标准的 `django.contrib.auth.logout()` 函数实际上所做的内容比这个要多一点以防止意外的数据泄露。它调用的 `request.session` 的 `flush()` 方法。我们使用这个例子来演示如何利用会话对象来工作，而不是一个完整的 `logout()` 实现。

设置测试的Cookie

为了方便，Django 提供一个简单的方法来测试用户的浏览器时候接受Cookie。只需在一个视图中调用 `request.session` 的 `set_test_cookie()` 方法，并在接下来的视图中调用 `test_cookie_worked()` —— 不是在同一个视图中调用。

由于Cookie的工作方式，在 `set_test_cookie()` 和 `test_cookie_worked()` 之间这种笨拙的分离是必要的。当你设置一个Cookie，直到浏览器的下一个请求你不可能真实知道一个浏览器是否接受了它。

使用 `delete_test_cookie()` 来自己清除测试的Cookie是一个很好的实践。请在你已经验证测试的Cookie 已经工作后做这件事。

下面是一个典型的使用示例：

```
def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try a
gain.")
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

在视图外使用会话

> 注
>
> 这一节中的示例直接从 `django.contrib.sessions.backends.db` 中导 入`SessionStore` 对象。在你的代码中，你应该从 `SESSION_ENGINE` 指定的会话引擎中导入 `SessionStore` ，如下所示：

```
>>> from importlib import import_module
>>> from django.conf import settings
>>> SessionStore = import_module(settings.SESSION_ENGINE).Sessio
nStore
```

在视图的外面有一个API 可以使用来操作会话的数据：

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since datetimes are not seri
alizable in JSON.
>>> s['last_login'] = 1376587691
>>> s.save()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'

>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6cee
ad')
>>> s['last_login']
1376587691
```

为了减缓会话固话攻击，不存在的会话的键将重新生成：

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore(session_key='no-such-session-here')
>>> s.save()
>>> s.session_key
'ff882814010ccbc3c870523934fee5a2'
```

如果你使用的是 `django.contrib.sessions.backends.db` 后端，每个会话只是一个普通的Django 模型。 `Session` 模型定义在 `django/contrib/sessions/models.py` 中。因为它是一个普通的模型，你可以使用普通的Django 数据库API 来访问会话：

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead
')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

注意，你需要调用 `get_decoded()` 以获得会话的字典。这是必需的，因为字典是以编码后的格式保存的：

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

会话何时保存

默认情况下，Django 只有在会话被修改时才会保存会话到数据库中 —— 即它的字典中的任何值被赋值或删除时：

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

上面例子的最后一种情况，我们可以通过设置会话对象的 `modified` 属性显式地告诉会话对象它已经被修改过：

```
request.session.modified = True
```

若要修改这个默认的行为，可以设置 `SESSION_SAVE_EVERY_REQUEST` 为 `True` 。当设置为 `True` 时，Django 将对每个请求保存会话到数据库中。

注意会话的Cookie 只有在一个会话被创建或修改后才会发送。如果 `SESSION_SAVE_EVERY_REQUEST` 为 `True` ，会话的Cookie 将在每个请求中发送。

类似地，会话Cookie 的 `expires` 部分在每次发送会话Cookie 时更新。

如果响应的状态码时500，则会话不会被保存。

浏览器时长的会话 VS. 持久的会话

你可以通过 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置来控制会话框架使用浏览器时长的会话，还是持久的会话。

默认情况下，`SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `False` ，表示会话的Cookie 保存在用户的浏览器中的时间为 `SESSION_COOKIE_AGE` 。如果你不想让大家每次打开浏览器时都需要登录时可以这样使用。

如果 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `True` ，Django 将使用浏览器时长的Cookie —— 用户关闭他们的浏览器时立即过期。如果你想让大家在每次打开浏览器时都需要登录时可以这样使用。

这个设置是一个全局的默认值，可以通过显式地调 `request.session` 的 `set_expiry()` 方法来覆盖，在上面的在视图中使用会话中有描述。

> 注
>
> 某些浏览器（例如Chrome）提供一种设置，允许用户在关闭并重新打开浏览器后继续使用会话。在某些情况下，这可能干扰 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置并导致会话在浏览器关闭后不会过期。在测试启用 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置的Django 应用时请注意这点。

清除存储的会话

随着用户在你的网站上创建新的会话，会话数据可能会在你的会话存储仓库中积累。如果你正在使用数据库作为后端， `django_session` 数据库表将持续增长。如果你正在使用文件作为后端，你的临时目录包含的文件数量将持续增长。

要理解这个问题，考虑一下数据库后端发生的情况。当一个用户登入时，Django 添加一行到 `django_session` 数据库表中。每次会话数据更新时，Django 将更新这行。如果用户手工登出，Django 将删除这行。但是如果该用户不登出，该行将永远不会删除。以文件为后端的过程类似。

Django 不提供自动清除过期会话的功能。因此，定期地清除会话是你的任务。Django 提供一个清除用的管理命令来满足这个目的： `clearsessions` 。建议定义调用这个命令，例如作为一个每天运行的Cron 任务。

注意，以缓存为后端不存在这个问题，因为缓存会自动删除过期的数据。以cookie 为后端也不存在这个问题，因为会话数据通过用户的浏览器保存。

设置

一些Django 设置 让你可以控制会话的行为：

- SESSION_CACHE_ALIAS
- SESSION_COOKIE_AGE
- SESSION_COOKIE_DOMAIN
- SESSION_COOKIE_HTTPONLY
- SESSION_COOKIE_NAME
- SESSION_COOKIE_PATH
- SESSION_COOKIE_SECURE

- SESSION_ENGINE
- SESSION_EXPIRE_AT_BROWSER_CLOSE
- SESSION_FILE_PATH
- SESSION_SAVE_EVERY_REQUEST

会话的安全

一个站点下的子域名能够在客户端为整个域名设置Cookie。如果子域名不收信任的用户控制且允许来自子域名的Cookie，那么可能发生会话固定。

例如，一个攻击者可以登录 `good.example.com` 并为他的账号获取一个合法的会话。如果该攻击者具有 `bad.example.com` 的控制权，那么他可以使用这个域名来发送他的会话ID给你，因为子域名允许在 `*.example.com` 上设置Cookie。当你访问 `good.example.com` 时，你将被登录成攻击者而没有注意到并输入你的敏感的个人信息（例如，信用卡信息）到攻击者的账号中。

另外一个可能的攻击是，如果 `good.example.com` 设置它的 `SESSION_COOKIE_DOMAIN` 为" `.example.com` " ，这将导致来自该站点的会话Cookie 被发送到 `bad.example.com` 。

技术细节

- 当使用 `JSONSerializer` 时，会话字典接收任何可json 序列化的值，当使用 `PickleSerializer` 时接收任何pickleable 的Python对象。更多信息参见 `pickle` 模块。
- 会话数据存储在数据中名为 `django_session` 的表中。
- Django 只发送它需要的Cookie。如果你没有设置任何会话数据，它将不会发送会话Cookie。

URL 中的会话ID

Django 会话框架完全地、唯一地基于Cookie。它不像PHP一样，实在没办法就把会话的ID放在URL 中。这是一个故意的设计。这个行为不仅使得URL变得丑陋，还使得你的网站易于受到通过"Referer" 头部窃取会话ID的攻击。

> 译者：Django 文档协作翻译小组，原文：Sessions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Sitemap 框架

Django 自带了一个高级的网站地图创建框架， 这使得创建XML格式的网站地图 变得容易。

概述

一个站点地图是一个在你网站上的用来告诉搜索引擎你的页面更新的多频繁和某些页面在你的网站中的重要关系的索引的XML文件此信息有助于搜索引擎为您的网站编制索引。

Django sitemap 框架通过让你在 Python 代码中表达此信息，自动创建此 XML 文件。

它的工作原理很像 Django 的联合框架。为了创建网站地图，只需编写 `Sitemap` 类，并在 *URLconf* 中指向该类。

安装

安装网站地图APP的步骤如下：

1. 在 `INSTALLED_APPS` 设置中添加 `'django.contrib.sitemaps'` .
2. 确认你的 `TEMPLATES` 设置包含 `DjangoTemplates` 后端，并将 `APP_DIRS` 选项设置为 `True` . 当然默认值就是这样的，只有当你曾经修改过这些设置，才需要修改这个配置。
3. 确认你已经安装 `sites framework` .

(注意: 网站地图APP并不安装任何数据库表 。需要修改 `INSTALLED_APPS` 的唯一原因是，以便 `Loader()` 模板加载器可以找到默认模板。)

初始化

`views.``sitemap` (*request*, *sitemaps*, *section=None*, *template_name='sitemap.xml'*, *content_type='application/xml'*)

为了在你的Django网站激活网站地图生成功能, 请把以下代码添加 *URLconf*:

```
from django.contrib.sitemaps.views import sitemap

url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
    name='django.contrib.sitemaps.views.sitemap')
```

当客服端访问 `/sitemap.xml` 时，这将告诉Django生成一个网站地图。

网站地图的文件名并不重要，重要的是文件的位置。搜索引擎只会索引网站的当前URL 层级及下属层级。例如，如果 `sitemap.xml` 位于根目录中，它可能会引用网站中的任何 URL。但是，如果站点地图位于 `/content/sitemap.xml` ，则它只能引用以 `/content/` 开头的网址。

网站地图视图需要一个额外的必需参数： `{'sitemaps'： sitemaps}` 。 `sitemaps` 应是一个字典，将小节的标签（例如 `blog` 或 `news` ）映射到其 `Sitemap` 类（例如， `BlogSitemap` 或 `NewsSitemap` ）。它也可以映射到 `Sitemap` 类的实例（例如， `BlogSitemap（some_var）` ）。

Sitemap 类

`Sitemap` 类是一个简单的 Python 类，表示站点地图中"一部分"条目。例如，一个 `Sitemap` 类可以表示 Weblog 的所有条目，而另一个可以表示事件日历中的所有事件。

在最简单的情况下，所有这些部分都集中到一个 `sitemap.xml` 中，但也可以使用框架，为每个部分生成一个站点地图索引，它引用单个站点地图文件。（请参阅下面的创建网站地图索引。）

`Sitemap` 类必须继承自 `django.contrib.sitemaps.Sitemap` 。它们可以位于你的代码库中的任何地方。

一个简单示例

假设你有一个博客系统，拥有 `Entry` 模型，并且您希望站点地图包含指向各个博客条目的所有链接。以下是您的Sitemap类别的外观：

```
from django.contrib.sitemaps import Sitemap
from blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

注意：

- `changefreq` 和 `priority` 分别是对应于 `<changefreq>` 和 `<priority>` 它们可以作为函数调用，例如这个例子中的 `lastmod` 。
- `items()` 只是一个返回对象列表的方法。返回的对象将传递给与网站地图属性（ `location` ， `lastmod` ， `changefreq` 和 `priority`
- `lastmod` 应返回 Python `datetime` 对象。
- 在此示例中没有 `location` 方法，但你可以提供此方法来指定对象的 URL。默认情况下， `location()` 在每个对象上调用 `get_absolute_url()` 并返回结果。

Sitemap 类参考

class `Sitemap` [source]

`Sitemap` 类可以定义以下方法/属性：

`items` [source]

必需。返回对象列表的方法。框架不关心它们是什么类型；所有重要的是这些对象被传递到 `location()` ， `lastmod()` ， `changefreq()` 和 `priority()` 方法。

`location` [source]

可选的. 进入一个方法或属性

如果它是一个方法, 它应该为 `items()` 返回的对象返回绝对路径.

如果它是一个属性，它的值应该是一个字符串，表示 `items()` 返回的每个对象的绝对路径。

在这两种情况下，“绝对路径”表示不包含协议或域的URL。例子：

- 好： `'/foo/bar/'`
- 错误： `'example.com/foo/bar/'`
- 错误： `'http://example.com/foo/bar/'`

如果未提供 `location` ，框架将调用 `items()` 返回的每个对象上的 `get_absolute_url()` 方法。

要指定 `'http'` 之外的协议，请使用 `protocol` 。

`lastmod`

可选。方法或属性。

如果它是一个方法，它应该接受一个参数 - 由 `items()` 返回的对象，并返回对象的最后修改日期/时间，如Python `datetime.datetime`

If it’s an attribute, its value should be a Python `datetime.datetime` object representing the last-modified date/time for *every* object returned by `items()` .

New in Django 1.7.

如果网站地图中的所有项目都有 `lastmod` ，则由 `views.sitemap()` 生成的网站地图会有 `Last-Modified` 最新 `lastmod` 。您可以激活 `ConditionalGetMiddleware` ，使Django对具有 `If-Modified-Since` 标头的请求作出适当响应，如果没有更改，则会阻止发送站点地图。

`changefreq`

可选。方法或属性。

如果它是一个方法，它应该接受一个参数 - 由 `items()` 返回的对象 - 并将对象的更改频率作为Python字符串返回。

如果是属性，则其值应为表示 `items()` 返回的每个对象的更改频率的字符串。

不管您使用方法还是属性， `changefreq` 的可能值为：

- `'always'`
- `'hourly'`
- `'daily'`
- `'weekly'`
- `'monthly'`
- `'yearly'`
- `'never'`

`priority`

可选。方法或属性。

如果它是一个方法，它应该接受一个参数 - 由 `items()` 返回的对象 - 并返回对象的优先级，如字符串或浮点数。

如果它是一个属性，它的值应该是一个字符串或浮动，表示 `items()` 返回的每个对象的优先级。

`priority` 的示例值： `0.4` ， `1.0` 。页面的默认优先级为 `0.5` 。有关详情，请参阅sitemaps.org文档。

`protocol`

可选。

此属性定义网站地图中的网址的协议（ `'http'` 或 `'https'` ）。如果未设置，则使用请求站点地图的协议。如果Sitemap是在请求的上下文之外构建的，则默认为 `'http'` 。

`limit`

可选。

此属性定义网站地图的每个网页上包含的最大网址数。其值不应超过 `50000` 的默认值，这是Sitemaps协议中允许的上限。

`i18n`

New in Django 1.8.

可选。

一个boolean属性，用于定义是否应使用您的所有 `LANGUAGES` 生成此网站地图的网址。默认值为 `False` 。

快捷方式

对于常见的情况，Sitemap框架提供了一些方便的类：

class `FlatPageSitemap` [source]

自1.8版起已弃用：请改用 `django.contrib.flatpages.sitemaps.FlatPageSitemap` 。

`django.contrib.sitemaps.FlatPageSitemap` 类查看为当前 `SITE_ID` 定义的所有公开可见的 `flatpages` （请参阅 `sites documentation` ），并在站点地图中创建一个条目。这些条目仅包含 `location` 属性 - 不是 `lastmod` ， `changefreq` 或 `priority` 。

class `GenericSitemap` [source]

`django.contrib.sitemaps.GenericSitemap` 类允许您通过传递一个必须至少包含 `queryset` 条目的字典来创建站点地图。此查询集将用于生成站点地图的项目。它还可以具有 `date_field` 条目，其指定从 `queryset` 检索的对象的日期字段。这将用于生成的站点地图中的 `lastmod` 属性。您还可以将 `priority` 和 `changefreq` 关键字参数传递到 `GenericSitemap` 构造函数，以指定所有网址的这些属性。

例

以下是使用 `GenericSitemap` 的*URLconf*的示例：

```
from django.conf.urls import url
from django.contrib.sitemaps import GenericSitemap
from django.contrib.sitemaps.views import sitemap
from blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

urlpatterns = [
    # some generic view using info_dict
    # ...

    # the sitemap
    url(r'^sitemap\.xml$', sitemap,
        {'sitemaps': {'blog': GenericSitemap(info_dict, priority
=0.6)}},
        name='django.contrib.sitemaps.views.sitemap'),
]
```

静态视图的Sitemap

通常，您希望搜索引擎抓取工具索引既不是对象详细信息页面也不是平面页的视图。解决方案是在 `items` 中显式列出这些视图的网址名称，并在网站地图的 `location` 方法中调用 `reverse()` 。例如：

```
# sitemaps.py
from django.contrib import sitemaps
from django.core.urlresolvers import reverse

class StaticViewSitemap(sitemaps.Sitemap):
    priority = 0.5
    changefreq = 'daily'

    def items(self):
        return ['main', 'about', 'license']

    def location(self, item):
        return reverse(item)

# urls.py
from django.conf.urls import url
from django.contrib.sitemaps.views import sitemap

from .sitemaps import StaticViewSitemap
from . import views

sitemaps = {
    'static': StaticViewSitemap,
}

urlpatterns = [
    url(r'^$', views.main, name='main'),
    url(r'^about/$', views.about, name='about'),
    url(r'^license/$', views.license, name='license'),
    # ...
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
        name='django.contrib.sitemaps.views.sitemap')
]
```

创建网站地图索引

`views.``index` (*request*, *sitemaps*, *template_name='sitemap_index.xml'*, *content_type='application/xml'*, *sitemap_url_name='django.contrib.sitemaps.views.sitemap'*)

站点地图框架还能够创建引用单个站点地图文件的站点地图索引，您的 `sitemaps` 字典中定义的每个部分一个。唯一的区别是：

- 您在URLconf中使用两个视图：`django.contrib.sitemaps.views.index()` 和 `django.contrib.site`。
- `django.contrib.sitemaps.views.sitemap()` 视图应采用 `section` 关键字参数。

这里是上面的例子的相关URLconf行：

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^sitemap\.xml$', views.index, {'sitemaps': sitemaps}),
    url(r'^sitemap-(?P<section>.+)\.xml$', views.sitemap, {'site
maps': sitemaps}),
]
```

这将自动生成引用 `sitemap-flatpages.xml` 和 `sitemap-blog.xml` 的 `sitemap.xml` 文件。 `Sitemap` 类和 `sitemaps` 类别根本不会更改。

如果您的其中一个Sitemap包含超过50,000个网址，就应该建立索引档。在这种情况下，Django会自动对网站地图分页，索引会反映出来。

如果您不使用vanilla网站地图视图（例如，如果使用缓存装饰器包装），则必须为您的网站地图视图命名，并将 `sitemap_url_name` 传递到索引视图：

```
from django.contrib.sitemaps import views as sitemaps_views
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^sitemap\.xml$',
        cache_page(86400)(sitemaps_views.index),
        {'sitemaps': sitemaps, 'sitemap_url_name': 'sitemaps'}),
    url(r'^sitemap-(?P<section>.+)\.xml$',
        cache_page(86400)(sitemaps_views.sitemap),
        {'sitemaps': sitemaps}, name='sitemaps'),
]
```

模板定制

如果您希望为网站上可用的每个站点地图或站点地图索引使用不同的模板，您可以通过将 `template_name` 参数传递到 `sitemap` 和 `index` 视图：

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^custom-sitemap\.xml$', views.index, {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
    url(r'^custom-sitemap-(?P<section>.+)\.xml$', views.sitemap,
 {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
]
```

这些视图返回 `TemplateResponse` 实例，允许您在渲染之前轻松自定义响应数据。有关详细信息，请参阅*TemplateResponse documentation*。

上下文变量

当自定义 `index()` 和 `sitemap()` 视图的模板时，您可以依赖于以下上下文变量。

指数

变量 `sitemaps` 是每个站点地图的绝对网址列表。

Sitemap

变量 `urlset` 是应显示在网站地图中的网址列表。每个网址都会显示在 `Sitemap` 类中定义的属性：

- `changefreq`
- `item`
- `lastmod`
- `location`
- `priority`

已为每个网址添加了 `item` 属性，以允许更灵活地自定义模板，例如Google新闻站点地图。假设Sitemap的 `items()` 会传回含有 `publication_data` 和 `tags` 的项目清单，系统就会产生Google新闻相容的Sitemap：

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset
  xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:news="http://www.google.com/schemas/sitemap-news/0.9">
{% spaceless %}
{% for url in urlset %}
  <url>
    <loc>{{ url.location }}</loc>
    {% if url.lastmod %}<lastmod>{{ url.lastmod|date:"Y-m-d" }}<
/lastmod>{% endif %}
    {% if url.changefreq %}<changefreq>{{ url.changefreq }}</cha
ngefreq>{% endif %}
    {% if url.priority %}<priority>{{ url.priority }}</priority>
{% endif %}
    <news:news>
      {% if url.item.publication_date %}<news:publication_date>{
{ url.item.publication_date|date:"Y-m-d" }}</news:publication_da
te>{% endif %}
      {% if url.item.tags %}<news:keywords>{{ url.item.tags }}</
news:keywords>{% endif %}
    </news:news>
   </url>
{% endfor %}
{% endspaceless %}
</urlset>
```

正在Ping Google

你可能希望在 Sitemap 更改时“ping”Google，以便让其重新索引你的网站。sitemaps框架提供了一个函数： `django.contrib.sitemaps.ping_google()` 。

`ping_google` ()[source]

`ping_google()` 使用可选参数 `sitemap_url` ，该参数应为网站站点地图的绝对路径（例如 `'/sitemap.xml'` 如果未提供此参数，则 `ping_google()` 将尝试通过在URLconf中执行反向查找来确定您的站点地图。

`ping_google()` 如果无法确定您的站点地图网址，则会引发例外 `django.contrib.sitemaps.SitemapNotFound` 。

先注册Google！

只有您已经使用Google网站管理员工具注册了您的网站， `ping_google()` 命令才会生效。

调用 `ping_google()` 的一个有用方法是从模型的 `save()` 方法：

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
    def save(self, force_insert=False, force_update=False):
        super(Entry, self).save(force_insert, force_update)
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a variety
            # of HTTP-related exceptions.
            pass
```

然而，更有效的解决方案是从cron脚本或其他计划的任务调用 `ping_google()` 。该函数向Google的服务器发出HTTP请求，因此您可能不想在每次调用 `save()` 时引入该网络开销。

正在通过Google Ping `manage.py`

`django-admin ping_google`

将站点地图应用程序添加到您的项目后，您还可以使用 `ping_google` 管理命令ping Google：

```
python manage.py ping_google [/sitemap.xml]
```

静态文件应用

`django.contrib.staticfiles` 从你的应用（和其他你指定的地方）收集所有静态文件到同一个地方，这样产品就能很容易的被维护

看看这里

对于静态文件的应用和一些用法示例的介绍，请参阅管理静态文件（*CSS*，图像）. 如果你想知道如何部署静态文件, 请参阅 部署静态文件.

设置

查看*staticfiles settings*了解更多设置细节

- STATIC_ROOT
- STATIC_URL
- STATICFILES_DIRS
- STATICFILES_STORAGE
- STATICFILES_FINDERS

管理命令

`django.contrib.staticfiles` 公开三个管理命令

搜集静态文件

`django-admin collectstatic`

搜集静态文件到 `STATIC_ROOT` .

默认情况下，重复文件名以类似于模板分辨率工作原理的方式解析：将使用首先在指定位置之一找到的文件。如果您感到困惑， `findstatic` 命令可以帮助您显示找到的文件。

使用 `enabled finders` 搜索文件。默认值是查看由 `STATICFILES_DIRS` 中定义的所有位置以及在由 `INSTALLED_APPS` 设置指定的应用程序的 `'static'` 目录中定义的所有位置。

`collectstatic` 管理命令在每次运行后调用 `STATICFILES_STORAGE` 的 `post_process()` 方法，并传递管理所发现的路径列表命令。它还接收 `collectstatic` 的所有命令行选项。默认情况下，它由 `CachedStaticFilesStorage` 使用。

默认情况下，收集的文件从 `FILE_UPLOAD_PERMISSIONS` 接收权限，收集的目录从 `FILE_UPLOAD_DIRECTORY_PERMISSIONS` 接收权限。如果您希望对这些文件和/或目录使用不同的权限，可以暂存*static files storage classes*，并指定 `file_permissions_mode` 和/或 `directory_permissions_mode` 例如：

```
from django.contrib.staticfiles import storage

class MyStaticFilesStorage(storage.StaticFilesStorage):
    def __init__(self, *args, **kwargs):
        kwargs['file_permissions_mode'] = 0o640
        kwargs['directory_permissions_mode'] = 0o760
        super(MyStaticFilesStorage, self).__init__(*args, **kwar
gs)
```

然后将 `STATICFILES_STORAGE` 设置为 `'path.to.MyStaticFilesStorage'` 。

New in Django 1.7:

覆盖 `file_permissions_mode` 和 `directory_permissions_mode` 的功能是Django 1.7中的新功能。以前，文件权限始终使用 `FILE_UPLOAD_PERMISSIONS` 和始终使用的目录权限 `FILE_UPLOAD_DIRECTORY_PERMISSIONS` 。

一些常用的选项是：

`--noinput`

不要提示用户输入任何类型。

`-i` `<pattern>`

`--ignore` `<pattern>`

忽略与此glob样式模式匹配的文件或目录。使用多次忽略更多。

`-n`

`--dry-run`

除了修改文件系统之外，执行所有操作。

`-c`

`--clear`

在尝试复制或链接原始文件之前清除现有文件。

`-l`

`--link`

创建指向每个文件的符号链接，而不是复制。

`--no-post-process`

不要调用配置的 `STATICFILES_STORAGE` 存储后端的 `post_process()` 方法。

`--no-default-ignore`

不要忽略常见的私有glob样式模式 `'CVS'` ， `'.*'` 和 `'*~'` 。

有关选项的完整列表，请参阅命令自己的帮助，运行：

```
$ python manage.py collectstatic --help
```

findstatic

`django-admin findstatic`

使用已启用的查找器搜索一个或多个相对路径。

例如：

```
$ python manage.py findstatic css/base.css admin/js/core.js
Found 'css/base.css' here:
 /home/special.polls.com/core/static/css/base.css
 /home/polls.com/core/static/css/base.css
Found 'admin/js/core.js' here:
 /home/polls.com/src/django/contrib/admin/media/js/core.js
```

默认情况下，找到所有匹配的位置。要仅返回每个相对路径的第一个匹配，请使用 `--first` 选项：

```
$ python manage.py findstatic css/base.css --first
Found 'css/base.css' here:
 /home/special.polls.com/core/static/css/base.css
```

这是一个调试助手；它会显示给定路径将收集哪个静态文件。

通过将 `--verbosity` 标志设置为0，可以抑制额外的输出，只需获取路径名称：

```
$ python manage.py findstatic css/base.css --verbosity 0
/home/special.polls.com/core/static/css/base.css
/home/polls.com/core/static/css/base.css
```

另一方面，通过将 `--verbosity` 标志设置为2，可以获取所有搜索的目录：

```
$ python manage.py findstatic css/base.css --verbosity 2
Found 'css/base.css' here:
  /home/special.polls.com/core/static/css/base.css
  /home/polls.com/core/static/css/base.css
Looking in the following locations:
  /home/special.polls.com/core/static
  /home/polls.com/core/static
  /some/other/path/static
```

New in Django 1.7:

添加了搜索其目录的其他输出。

runserver

`django-admin runserver`

Overrides the core `runserver` command if the `staticfiles` app is `installed` and adds automatic serving of static files and the following new options.

`--nostatic`

使用 `--nostatic` 选项可以完全禁止使用应用程序提供静态文件。仅当应用位于项目的 `INSTALLED_APPS` 设置中时，此选项才可用。

用法示例：

```
django-admin runserver --nostatic
```

`--insecure`

使用 `--insecure` 选项强制使用应用程式提供静态档案，即使 `DEBUG` 设定为 `False` 通过使用此功能，您可以确认严重无效以及可能不安全。这只适用于本地开发，应从不用于生产，并且仅当应用程序位于项目的 `INSTALLED_APPS` 设置时可用。 `runserver` `--insecure` 不适用于 `CachedStaticFilesStorage` 。

用法示例：

```
django-admin runserver --insecure
```

存储

StaticFilesStorage

class `storage.``StaticFilesStorage`

使用 `STATIC_ROOT` 设置作为基本文件系统位置和 `STATIC_URL` 设置的 `FileSystemStorage` 存储后端的子类分别作为基本URL。

`storage.StaticFilesStorage.``post_process` (*paths*, ***options*)

此方法在每次运行后由 `collectstatic` 管理命令调用，并将找到的文件的本地存储和路径作为字典以及命令行选项传递。

`CachedStaticFilesStorage` 在幕后使用它来替换路径与它们的哈希对等体，并适当地更新缓存。

ManifestStaticFilesStorage

New in Django 1.7.

class `storage.``ManifestStaticFilesStorage`

`StaticFilesStorage` 存储后端的子类，通过将文件内容的MD5哈希附加到文件名来存储它处理的文件名。例如，文件 `css/styles.css` 也将另存为 `css/styles.55e7cbb9ba48.css` 。

此存储的目的是为了在一些页面仍然引用这些文件的情况下继续提供旧文件，例如。因为它们由您或第三方代理服务器缓存。此外，如果您希望将远期Expires标头应用于已部署的文件，以加快后续网页访问的加载时间，这将非常有帮助。

存储后端会自动使用缓存副本的路径（使用 `post_process()` 方法）替换保存的文件中与其他已保存文件匹配的路径。默认情况下，用于查找这些路径（ `django.contrib.staticfiles.storage.HashedFilesMixin.patterns` ）的正则表达式涵盖@import规则和url() / t3> 级联样式表的语句。例如， `'css/styles.css'` 文件带有内容

```
@import url("../admin/css/base.css");
```

将替换为调用 `ManifestStaticFilesStorage` 存储后端的 `url()` 方法，最终保存一个 `'css/styles.55e7cbb9ba48.css'` 具有以下内容：

```
@import url("../admin/css/base.27e20196a850.css");
```

要启用 `ManifestStaticFilesStorage` ，您必须确保满足以下要求：

- `STATICFILES_STORAGE` 设置设置为 `'django.contrib.staticfiles.storage.ManifestStaticFilesStorage`
- `DEBUG` 设置为 `False`
- 您可以使用 `staticfiles` `static` 模板标记来引用模板中的静态文件
- 您已使用 `collectstatic` 管理命令收集了所有静态文件

由于创建MD5哈希值会对运行时的网站造成负担，因此 `staticfiles` 会自动将所有已处理文件的哈希值映射存储在名为 `staticfiles.json` 。当您运行 `collectstatic` 管理命令时，会发生这种情况。

由于运行 `collectstatic` 的要求，在运行测试时，通常不应使用此存储器，因为 `collectstatic` 不作为正常测试设置的一部分运行。在测试期间，请确保 `STATICFILES_STORAGE` 设置设置为像 `'django.contrib.staticfiles.storage.StaticFilesStorage'` （默认值）。

`storage.ManifestStaticFilesStorage.``file_hash` (*name*, *content=None*)

创建文件的散列名称时使用的方法。需要返回给定文件名和内容的哈希值。默认情况下，它从内容的块计算MD5哈希，如上所述。随意覆盖此方法使用自己的哈希算法。

CachedStaticFilesStorage

class `storage.``CachedStaticFilesStorage`

`CachedStaticFilesStorage` 类似于 `ManifestStaticFilesStorage` 类，但使用Django的*caching framework*来存储处理文件的哈希名称，而不是静态清单文件 `staticfiles.json` 。这在您无权访问文件系统的情况下非常有用。

如果要覆盖存储使用的高速缓存后端的某些选项，只需在名为 `'staticfiles'` 的 `CACHES` 设置中指定自定义条目即可。它会回到使用 `'default'` 缓存后端。

模板标签

静态的

使用配置的 `STATICFILES_STORAGE` 存储来为给定的相对路径创建完整的网址，例如：

```
{% load static from staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

上一个示例等于使用 `"images/hi.jpg"` 调用 `STATICFILES_STORAGE` 实例的 `url` 方法。这在使用非本地存储后端部署文件时特别有用，如*Serving static files from a cloud service or CDN*提供静态文件中所述。

如果您希望在不显示静态网址的情况下检索静态网址，则可以使用略有不同的调用：

```
{% load static from staticfiles %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}" alt="Hi!" />
```

查找模块

`staticfiles` 查找器具有 `searched_locations` 属性，它是查找器搜索的目录路径的列表。用法示例：

```
from django.contrib.staticfiles import finders

result = finders.find('css/base.css')
searched_locations = finders.searched_locations
```

New in Django 1.7.

已添加 `searched_locations` 属性。

其他帮助

除了 `staticfiles` 应用程序之外还有一些其他助手可以使用静态文件：

- `django.template.context_processors.static()` 上下文处理器，将 `STATIC_URL` 添加到用 `RequestContext` 上下文渲染的每个模板上下文。
- 内置模板标记 `static` ，它接受一个路径，并使用静态前缀 `STATIC_URL` 将其链接。
- 内置模板标记 `get_static_prefix` 用于将模板变量填充为静态前缀 `STATIC_URL` 以用作变量或直接。
- 类似的模板标签 `get_media_prefix` ，其工作方式类似于 `get_static_prefix` ，但使用 `MEDIA_URL` 。

静态文件开发视图

静态文件工具主要用于帮助将静态文件成功部署到生产环境中。这通常意味着一个单独的，专用的静态文件服务器，这在开发本地时是很麻烦的开销。因此， `staticfiles` 应用程序附带了一个快速和脏的帮助视图，您可以使用它在开发中本地提供文件。

`views.``serve` (*request*, *path*)

此视图函数在开发中提供静态文件。

警告

此视图仅在 `DEBUG` 为 `True` 时有效。

这是因为此视图严重无效，可能不安全。这只适用于本地开发，应从不用于生产。

Changed in Django 1.7:

当 `DEBUG` 为 `False` 时，此视图现在将引发 `Http404` 异常而不是 `ImproperlyConfigured` 。

注意

要猜测提供的文件的内容类型，此视图依赖于来自Python标准库的 `mimetypes` 模块，该模块本身依赖于底层平台的映射文件。如果您发现此视图没有为特定文件返回正确的内容类型，则很可能是平台的地图文件需要更新。这可以通过在Debian发行版上安装或更新Red Hat发行版上的 `mailcap` 软件包或 `mime-support` 来实现。

此视图由 `runserver` 自动启用（ `DEBUG` 设置为 `True` ）。要使用不同本地开发服务器的视图，请将以下代码段添加到主URL配置的末尾：

```
from django.conf import settings
from django.contrib.staticfiles import views

if settings.DEBUG:
    urlpatterns += [
        url(r'^static/(?P<path>.*)$', views.serve),
    ]
```

注意，模式（ `r'^ static /'` ）的开头应该是你的 `STATIC_URL` 设置。

由于这有点麻烦，还有一个帮助函数，将为你做这个：

`urls.``staticfiles_urlpatterns` ()

这将返回正确的URL模式，用于将静态文件提供给已定义的模式列表。使用它像这样：

```
from django.contrib.staticfiles.urls import staticfiles_urlpatte
rns

# ... the rest of your URLconf here ...

urlpatterns += staticfiles_urlpatterns()
```

这将检查您的 `STATIC_URL` 设置，并将视图连接到相应的静态文件。不要忘记适当地设置 `STATICFILES_DIRS` 设置，让 `django.contrib.staticfiles` 知道除了应用程序目录中的文件之外还要查找文件的位置。

警告

只有 `DEBUG` 设置为 `True` ，并且 `STATIC_URL` 设置不为空和完整的URL路径，比如 `http://static.example.com/` ，这个帮助功能才会工作。

这是因为该视图 效率不高 并且很可能 不安全. 该视图仅适用于本地开发 ，而不应该用于项目实际生产环境.

Specialized test case to support ‘live testing’

class `testing.``StaticLiveServerTestCase`

这个单元测试TestCase子类扩展 `django.test.LiveServerTestCase` 。

就像它的父类，你可以使用它来编写测试，涉及运行测试中的代码，并使用 HTTP 测试工具（例如 Selenium，PhantomJS 等），因为它需要同时发布静态素材。

但是考虑到它使用了上面描述的 `django.contrib.staticfiles.views.serve()` 视图，它可以在测试执行时透明地覆盖由 `staticfiles` finders。这意味着您不需要在测试设置之前或作为测试设置的一部分运行 `collectstatic` 。

New in Django 1.7:

`StaticLiveServerTestCase` 是Django 1.7中的新功能。以前，它的功能由 `django.test.LiveServerTestCase` 提供。

验证器

编写验证器

验证器是一个可调用的对象，它接受一个值，并在不符合一些规则时抛出 `ValidationError` 异常。验证器有助于在不同类型的字段之间重复使用验证逻辑。

例如，这个验证器只允许偶数：

```
from django.core.exceptions import ValidationError

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError('%s is not an even number' % value
)
```

你可以通过字段的 `validators` 参数将它添加到模型字段中：

```
from django.db import models

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])
```

由于值在验证器运行之前会转化为Python，你可以在表单上使用相同的验证器：

```
from django import forms

class MyForm(forms.Form):
    even_field = forms.IntegerField(validators=[validate_even])
```

你也可以使用带有 `__call__()` 方法的类，来实现更复杂或可配置的验证器。例如， `RegexValidator` 就用了这种技巧。如果一个基于类的验证器用于 `validators` 模型字段的选项，你应该通过添加*deconstruct()* 和 `__eq__()` 方法确保它可以被迁移框架序列化。

验证器如何运行

关于验证器如何在表单中运行，详见表单验证 。关于它们如何在模型中运行，详见 验证对象。要注意验证器不会在你保存模型时自动运行，但是如果你使用 `ModelForm` ，它会在任何你表单包含的字段上运行你的验证器。关于模型验证

器如何和表单交互，详见*ModelForm* 文档。

内建的验证器

`django.core.validators` 模块包含了一系列的可调用验证器，用于模型和表单字段。它们在内部使用，但是也可以用在你自己的字段上。它们可以用在 `field.clean()` 方法之外，或者代替它。

RegexValidator

class `RegexValidator` ([*regex=None*, *message=None*, *code=None*, *inverse_match=None*, *flags=0*])[source]

Parameters:

- **regex** – 如果不是 `None` 则覆写 `regex` 。可以是一个正则表达式字符串，或者预编译的正则表达式对象。
- **message** – 如果不是 `None` ，则覆写 `message` 。
- **code** – 如果不是 `None` ，则覆写 `code` 。
- **inverse_match** – 如果不是 `None` ，则覆写 `inverse_match` 。
- **flags** – 如果不是 `None` ，则覆写 `flags` 。这种情况下， `regex` ，必须是正则表达式字符串，否则抛出 `TypeError` 异常。

`regex`

用于搜索提供的 `value` 的正则表达式，或者是预编译的正则表达式对象。通常在找不到匹配时抛出带有 `message` 和 `code` 的 `ValidationError` 异常。这一标准行为可以通过设置 `inverse_match` 为 `True` 来反转，这种情况下，如果找到匹配则抛出 `ValidationError` 异常。通常它会匹配任何字符串（包括空字符串）。

`message`

验证失败时 `ValidationError` 所使用的错误信息。默认为 `"Enter a valid value"` 。

`code`

验证失败时 `ValidationError` 所使用的错误代码。默认为 `"invalid"` 。

`inverse_match`

New in Django 1.7.

`regex` 的匹配模式。默认为 `False` 。

`flags`

New in Django 1.7.

编译正则表达式字符串 `regex` 时所用的标识。如果 `regex` 是预编译的正则表达式，并且覆写了 `flags` ，会产生 `TypeError` 异常。默认为 *0*。

EmailValidator

class `EmailValidator` ([*message=None*, *code=None*, *whitelist=None*])[source]

Parameters:

- **message** – 如果不是 `None` ，则覆写 `message` 。
- **code** – 如果不是 `None` ，则覆写 `code` 。
- **whitelist** – 如果不是 `None` ，则覆写 `whitelist` 。

`message`

验证失败时 `ValidationError` 所使用的错误信息。默认为 `"Enter a valid email address"` 。

`code`

验证失败时 `ValidationError` 所使用的错误代码。默认为 `"invalid"` 。

`whitelist`

所允许的邮件域名的白名单。通常，正则表达式(`domain_regex` 属性) 用于验证 @ 符号后面的任何东西。但是，如果这个字符串在白名单里，就可以通过验证。如果没有提供，默认的白名单是 `['localhost']` 。其它不包含点符号的域名不能通过验证，所以你需要按需将它们添加进白名单。

URLValidator

class `URLValidator` ([*schemes=None*, *regex=None*, *message=None*, *code=None*])[source]

`RegexValidator` 确保一个值看起来像是URL，并且如果不是的话产生 `'invalid'` 错误代码。

回送地址以及保留的IP空间被视为有效。同时也支持字面的IPv6地址 (**RFC 2732**) 以及unicode域名。

除了父类 `RegexValidator` 的可选参数之外， `URLValidator` 接受一个额外的可选属性：

`schemes`

需要验证的URL/URI模式列表。如果没有提供，默认为 `['http', 'https', 'ftp', 'ftps']` 。IANA 网站提供了 有效的URI模式的完整列表作为参考。

Changed in Django 1.7:

添加了可选的 `schemes` 属性。

Changed in Django 1.8:

添加了对IPv6 地址, unicode 域名, 以及含有验证信息的URL的支持。

validate_email

`validate_email`

一个不带有任何自定义的 `EmailValidator` 实例。

validate_slug

`validate_slug`

一个 `RegexValidator` 实例，确保值只含有字母、数字、下划线和连字符。

validate_ipv4_address

`validate_ipv4_address`

一个 `RegexValidator` 的实例，确保值是IPv4地址。

validate_ipv6_address

`validate_ipv6_address` [source]

使用 `django.utils.ipv6` 来检查是否是 IPv6 地址。

validate_ipv46_address

`validate_ipv46_address` [source]

使用 `validate_ipv4_address` 和 `validate_ipv6_address` 值是有效的 IPv4 或 IPv6 地址。

validate_comma_separated_integer_list

`validate_comma_separated_integer_list`

一个 `RegexValidator` 的实例，确保值是整数的逗号分隔列表。

MaxValueValidator

class `MaxValueValidator` (*max_value*, *message=None*)[source]

如果 `value` 大于 `max_value` ，抛出带有 `'max_value'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

MinValueValidator

class `MinValueValidator` (*min_value*, *message=None*)[source]

如果 `value` 小于 `min_value` ，抛出带有 `'min_value'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

MaxLengthValidator

class `MaxLengthValidator` (*max_length*, *message=None*)[source]

如果 `value` 的长度大于 `max_length` ，抛出带有 `'max_length'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

MinLengthValidator

class `MinLengthValidator` (*min_length*, *message=None*)[source]

如果 `value` 的长度小于 `min_length` ，抛出带有 `'min_length'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

> 译者：Django 文档协作翻译小组，原文：Data validation。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

其它核心功能

学习django框架的其他核心功能：

按需内容处理

HTTP客户端可能发送一些协议头来告诉服务端它们已经看过了哪些资源。这在获取网页（使用HTTP `GET` 请求）时非常常见，可以避免发送客户端已经获得的完整数据。然而，相同的协议头可用于所有HTTP方法(`POST` , `PUT` , `DELETE` ,以及其它)。

对于每一个Django从视图发回的页面（响应），都会提供两个HTTP协议头：`ETag` 和 `Last-Modified` 。这些协议头在HTTP响应中是可选的。它们可以由你的视图函数设置，或者你可以依靠 `CommonMiddleware` 中间件来设置 `ETag` 协议头。

当你的客户端再次请求相同的资源时，它可能会发送 If-modified-since 或者If-unmodified-since的协议头，包含之前发送的最后修改时间；或者 If-match 或If-none-match协议头，包含之前发送的 `ETag` 。如果页面的当前版本匹配客户端发送的 `ETag` ，或者如果资源没有被修改，会发回304状态码，而不是一个完整的回复，告诉客户端没有任何修改。根据协议头，如果页面被修改了，或者不匹配客户端发送的 `ETag` ，会返回412（先决条件失败，Precondition Failed）状态码。

当你需要更多精细化的控制时，你可以使用每个视图的按需处理函数。

Changed in Django 1.8:

向按需视图处理添加 `If-unmodified-since` 协议头的支持

The `condition`

有时（实际上是经常），你可以创建一些函数来快速计算出资源的ETag值或者最后修改时间，并不需要执行构建完整视图所需的所有步骤。Django可以使用这些函数来为视图处理提供一个“early bailout”的选项。来告诉客户端，内容自从上次请求并没有任何改动。

这两个函数作为参数传递到 `django.views.decorators.http.condition` 装饰器中。这个装时期使用这两个函数（如果你不能既快又容易得计算出来，你只需要提供一个）来弄清楚是否HTTP请求中的协议头匹配那些资源。如果它们不匹配，会生成资源的一份新的副本，并调用你的普通视图。

`condition` 装饰器的签名为i：

```
condition(etag_func=None, last_modified_func=None)
```

计算ETag的最后修改时间的两个函数，会以相同的顺序传入 `request` 对象和相同的参数，就像它们封装的视图函数那样。 `last_modified_func` 函数应该返回一个标准的datetime值，它制订了资源修改的最后时间，或者资源不存在为 `None` 。

传递给 `etag` 装饰器的函数应该返回一个表示资源Etag的字符串，或者资源不存在时为 `None` 。

用一个例子可以很好展示如何使用这一特性。假设你有这两个模型，表示一个简单的博客系统：

```
import datetime
from django.db import models

class Blog(models.Model):
    ...

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    published = models.DateTimeField(default=datetime.datetime.n
ow)
    ...
```

如果头版展示最后的博客文章，仅仅在你添加新文章的时候修改，你可以非常快速地计算出最后修改时间。你需要这个博客每一篇文章的最后 `发布` 日期。实现它的一种方式是：

```
def latest_entry(request, blog_id):
    return Entry.objects.filter(blog=blog_id).latest("published"
).published
```

接下来你可以使用这个函数，来为你的头版视图事先探测未修改的页面：

```
from django.views.decorators.http import condition

@condition(last_modified_func=latest_entry)
def front_page(request, blog_id):
    ...
```

只计算一个值的快捷方式

一个普遍的原则是，如果你提供了计算 ETag和最后修改时间的函数，你应该这样做：你并不知道HTTP客户端会发给你哪个协议头，所以要准备好处理两种情况。但是，有时只有二者之一容易计算，并且Django只提供给你计算ETag或最后修改日期的装饰器。

`django.views.decorators.http.etag` 和 `django.views.decorators.http.last_modified` 作为 `condition` 装饰器，传入相同类型的函数。他们的签名是：

```
etag(etag_func)
last_modified(last_modified_func)
```

我们可以编写一个初期的示例，它仅仅使用最后修改日期的函数，使用这些装饰器之一：

```
@last_modified(latest_entry)
def front_page(request, blog_id):
    ...
```

...或者：

```
def front_page(request, blog_id):
    ...
front_page = last_modified(latest_entry)(front_page)
```

Use `condition`

如果你想要测试两个先决条件，把 `etag` 和 `last_modified` 装饰器链到一起看起来很不错。但是，这会导致不正确的行为：

```
# Bad code. Don't do this!
@etag(etag_func)
@last_modified(last_modified_func)
def my_view(request):
    # ...

# End of bad code.
```

第一个装饰器不知道后面的任何事情，并且可能发送“未修改”的响应，即使第二个装饰器会处理别的事情。 `condition` 装饰器同时更使用两个回调函数，来弄清楚哪个是正确的行为。

使用带有其它HTTP方法的装饰器

`condition` 装饰器不仅仅对 `GET` 和 `HEAD` 请求有用（ `HEAD` 请求在这种情况下和 `GET` 相同）。它也可以用于为 `POST` ，`PUT` 和 `DELETE` 请求提供检查。在这些情况下，不是要返回一个“未修改（not modified，314）”的响应，而是要告诉服务端，它们尝试修改的资源在此期间被修改了。

例如，考虑以下客户端和服务端之间的交互：

1. 客户端请求 `/foo/` 。

2. 服务端回复一些带有 `"abcd1234"` ETag的内容。
3. 客户端发送HTTP `PUT` 请求到 `/foo/` 来更新资源。同时也发送了 `If-Match: "abcd1234"` 协议头来指定尝试更新的版本。
4. 服务端检查是否资源已经被修改，通过和 `GET` 上所做的相同方式计算ETag（使用相同的函数）。如果资源 已经 修改了，会返回412状态码，意思是“先决条件失败（precondition failed）”。
5. 客户端在接收到412响应之后，发送 `GET` 请求到 `/foo/` ，来在更新之前获取内容的新版本。

重要的事情是，这个例子展示了在所有情况下，ETag和最后修改时间值都采用相同函数计算。实际上，你 应该 使用相同函数，以便每次都返回相同的值。

使用中间件按需处理来比较

你可能注意到，Django已经通过 `django.middleware.http.ConditionalGetMiddleware` 和 `CommonMiddleware` .提供了简单和直接的 `GET` 的按需处理。这些中间件易于使用并且适用于多种情况，然而它们的功能有一些高级用法上的限制：

- 它们在全局上用于你项目中的所有视图。
- 它们不会代替你生成响应本身，这可能要花一些代价。
- 它们只适用于HTTP `GET` 请求。

在这里，你应该选择最适用于你特定问题的工具。如果你有办法快速计算出ETag和修改时间，并且如果一些视图需要花一些时间来生成内容，你应该考虑使用这篇文档描述的 `condition` 装饰器。如果一些都执行得非常快，坚持使用中间件在如果视图没有修改的条件下也会使发回客户端的网络流量也会减少。

> 译者：Django 文档协作翻译小组，原文：Conditional content processing。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Contenttypes 框架

Django 包含一个 `contenttypes` 应用，它可以追踪安装在你的Django 项目里的所有应用，并提供一个高层次的、通用的接口用于与你的模型进行交互。

概述

Contenttypes 的核心应用是 `ContentType` 模型，存在于 `django.contrib.contenttypes.models.ContentType`。`ContentType` 的实例表示并存储你的项目当中安装的应用的信息，并且新的 `ContentType` 实例每当新的模型安装时会自动创建。

`ContentType` 实例具有返回它们表示的模型类的方法，以及从这些模型查询对象的方法。`ContentType` 还有一个自定义的管理器用于添加方法来与 `ContentType` 工作，以及用于获得 `ContentType` 实例的特定模型。

你的模型和 `ContentType` 之间的关系还可以用于一个模型实例和任意一个已经安装的模型的实例建立“generic关联”。

安装Contenttypes 框架

Contenttypes 框架包含在 `django-admin startproject` 创建的默认的 `INSTALLED_APPS` 列表中，但如果你移除了它或者你手动创建 `INSTALLED_APPS` 列表，你可以通过添加 `'django.contrib.contenttypes'` 到你的 `INSTALLED_APPS` 设置中来启用它。

一般来说，安装Contenttypes 框架是个好主意。许多Django 的捆绑应用需要它：

- Admin 应用使用它来记录通过Admin 界面添加或更改每个对象的历史。
- Django 的 `authentication 框架` 用它来授用户权限给特殊的模型。

的 `ContentType`

class `ContentType` [source]

每一个 `ContentType` 实例有两个字段，共同来唯一描述一个已经安装的模型。

`app_label`

模型所在的应用的名称。这取自模型的 `app_label` 属性，并只包括应用的Python 导入路径的最后的部分。例如，"django.contrib.contenttypes"的 `app_label` 是"contenttypes"。

`model`

模型的类的名称。

此外，下面的属性是可用的：

`name` [source]

Contenttype 的人类可读的的名称。它取之于模型的 `verbose_name` 属性。

Changed in Django 1.8:

在Django 1.8 之前， `name` 属性是 `ContentType` 模型的一个字段。

让我们看看一个例子，看看它如何工作。如果你已经安装 `contenttypes` 应用，那么添加 `sites应用` 到你的 `INSTALLED_APPS` 设置并运行 `manage.py migrate` 来安装它，模型 `django.contrib.sites.models.Site` 将安装到你的数据库中。同时将创建 `ContentType` 的一个具有以下值的新实例：

- `app_label` 将设置为 `'sites'` （Python 路径"django.contrib.sites"的最后部分）。
- `model` 将设置为 `'site'` 。

方法 `ContentType`

每一个 `ContentType` 都有一些方法允许你用 `ContentType` 实例来到达它所代表的model, 或者从model取出对象:

`ContentType.``get_object_for_this_type` (***kwargs*)[source]

接收 `ContentType` 表示的模型所接收的查询参数，对该模型做 `一次get() 查询` ，然后返回相应的对象。

`ContentType.``model_class` ()[source]

返回此 `ContentType` 实例所表示的模型类。

例如，我们可以查找 `User` 模型的 `ContentType` ：

```
>>> from django.contrib.contenttypes.models import ContentType
>>> user_type = ContentType.objects.get(app_label="auth", model=
"user")
>>> user_type
<ContentType: user>
```

然后使用它来查询一个特定的 `User` ，或者访问 `User` 模型类：

```
>>> user_type.model_class()
<class 'django.contrib.auth.models.User'>
>>> user_type.get_object_for_this_type(username='Guido')
<User: Guido>
```

`get_object_for_this_type()` 和 `model_class()` 一起使用可以实现两个极其重要的功能：

1. 使用这些方法，你可以编写高级别的泛型代码，执行查询任何已安装的模型 —— 而不是导入和使用单一特定模型的类，可以通过 `app_label` 和 `model` 到 `ContentType` 在运行时查找，然后使用这个模型类或从它获取对象。
2. 你可以关联另一个模型到 `ContentType` 作为一种绑定它到特定模型类的方式，然后使用这些方法来获取对那些模型的访问。

几个Django 捆绑的应用利用后者的技术。例如，Django 的认证框架中的 权限系统 使用的 `Permission` 模型具有一个外键到 `ContentType` ；这允许 `Permission` 表示"可以添加博客条目"或"可以删除新闻故事"的概念。

的 `ContentTypeManager`

class `ContentTypeManager` [source]

`ContentType` 还具有自定义的管理器 `ContentTypeManager` ，它增加了下列方法：

`clear_cache` ()[source]

清除 `ContentType` 用于跟踪模型的内部缓存，它已为其创建 `ContentType` 实例。你可能不需要自己调用此方法；Django 将在它需要的时候自动调用。

`get_for_id` (*id*)[source]

通过ID查找 `ContentType` 。由于此方法使用与 `get_for_model()` 相同的共享缓存，建议使用这个方法而不是通常的 `ContentType.objects.get(pk=id)` 。

`get_for_model` (*model*[, *for_concrete_model=True*])[source]

接收一个模型类或模型的实例，并返回表示该模型的 `ContentType` 实例。 `for_concrete_model=False` 允许获取代理模型的 `ContentType` 。

`get_for_models` (**models*[, *for_concrete_models=True*])[source]

接收可变数目的模型类，并返回一个字典，将模型类映射到表示它们的 `ContentType` 实例。 `for_concrete_model=False` 允许获取代理模型的 `ContentType` 。

`get_by_natural_key` (*app_label*, *model*)[source]

返回由给定的应用标签和模型名称唯一标识的 `ContentType` 实例。这种方法的主要目的是为允许 `ContentType` 对象在反序列化期间通过自然键来引用。

`get_for_model()` 方法特别有用，当你知道你需要与 `ContentType` 交互但不想要去获取模型元数据以执行手动查找的麻烦：

```
>>> from django.contrib.auth.models import User
>>> user_type = ContentType.objects.get_for_model(User)
>>> user_type
<ContentType: user>
```

通用关系

在你的model添加一个外键到 `ContentType` 这将允许你更快捷的绑定自身到其他的model class，就像上述的 `Permission` model 一样。但是它非常有可能进一步的利用 `ContentType` 来实现真正的 generic (有时称之为多态) relationships 在models之间。

一个简单的例子是标记系统，它可能看起来像这样：

```
from django.db import models
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType

class TaggedItem(models.Model):
    tag = models.SlugField()
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey('content_type', 'object_i
d')

    def __str__(self):              # __unicode__ on Python 2
        return self.tag
```

一个普通的 `ForeignKey` 只能指向其他任意一个model，这是说 `TaggedItem` model用一个 `ForeignKey` 只能一个，并且也只能有这么一个model粗存tags。contenttypes application提供了一个特殊的字段(`GenericForeignKey`)避免了这个问题并且允许你和任何一个model建立关联关系。

class `GenericForeignKey` [source]

三个步骤建立 `GenericForeignKey` :

1. 给你的model设置一个 `ForeignKey` 字段到 `ContentType` . 一般命名为“content_type”.
2. 给你的model设置一个字段，用来储存你想要关联的model主键值。对于大多

数model,，这是一个 `PositiveIntegerField` 字段。并且通常命名为“object_id”.
3. 给你的model一个 `GenericForeignKey` 字段， 把1,2点提到的那两个字段的名词传给他。如果这两个字段名字分别为“content_type” 和 “object_id”, 你就可以省略他们 – `GenericForeignKey` 默认的会自动去查找这个两个命名的字段。

`for_concrete_model`

如果为 `False` ,那么字段将会涉及到proxy models（代理模型）。默认是 `True` . 这映射了 `for_concrete_model` 的参数到 `get_for_model()` .

`allow_unsaved_instance_assignment`

New in Django 1.8.

与 `ForeignKey.allow_unsaved_instance_assignment` 类似。

自1.7版起已弃用：此类过去在 `django.contrib.contenttypes.generic` 中定义。将从Django 1.9中删除从此旧位置导入的支持。

主键类型的兼容性。

“object_id” 字段并不总是相同的，这是由于储存在相关模型中的主键类型的关系,但是他们的主键值必须被转变为相同类型，这是通过 “object_id” 字段 的 `get_db_prep_value()` 方法。

例如, 如果你想要简历generic 关系到一个 `IntegerField` 或者 `CharField` 为主键的模型, 你可以使用 `CharField` 给 “object_id”字段，因为数字是可以被 `get_db_prep_value()` 转化为字母的。

为了更大的灵活性你也可以用 `TextField` ，一个没有限制最大长度的字段,，然而这可能给你的数据库带来巨大的影响。

这里没有一个一刀切的解决办法来应对哪个字段类型最好的问题。你应该估摸一下。哪些models 你想要关联，并根据此决定一个最佳方案。（废话。）

序列化 `ContentType` 的对象引用。

如果你想要序列化一个建立了 generic关系的model数据(for example, when generating `fixtures`) ，你应该用一个自然键来唯一的标识相关的 `ContentType` 对象。有关详细信息，请参阅*natural keys* 和 `dumpdata --natural-foreign` 。

这允许你像平时用 `ForeignKey` 类似的API来工作。每一个 `TaggedItem` 都将有一个 `content_object` 字段返回对象的关联，你也可以指定这个字段或者用它来创建一个 `TaggedItem` :

```
>>> from django.contrib.auth.models import User
>>> guido = User.objects.get(username='Guido')
>>> t = TaggedItem(content_object=guido, tag='bdfl')
>>> t.save()
>>> t.content_object
<User: Guido>
```

由于 `GenericForeignKey` 完成的方式问题,，你没有办法用这个字段直接执行数据库API，filters的操作。比如 (`filter()` and `exclude()` , for example) 。因为一个 `GenericForeignKey` 不是一个普通的字段对象t, 这些例子是不会工作的:

```
# This will fail
>>> TaggedItem.objects.filter(content_object=guido)
# This will also fail
>>> TaggedItem.objects.get(content_object=guido)
```

同样的, `GenericForeignKey` s 是不会出现在 `ModelForm` s.

反向通用关系

class `GenericRelation` [source]

自1.7版起已弃用：此类过去在 `django.contrib.contenttypes.generic` 中定义。将从Django 1.9中删除从此旧位置导入的支持。

`related_query_name`

New in Django 1.7.

默认情况下，相关对象返回到该对象的关系不存在。设置 `related_query_name` 来创建一个对象从关联对象返回到对象自身。这允许查询和筛选相关的对象。

如果你知道你最经常使用哪种型号的，你还可以添加一个“反向”的通用关系，以使其能附加一个附加的API。 例如：

```
class Bookmark(models.Model):
    url = models.URLField()
    tags = GenericRelation(TaggedItem)
```

`Bookmark` 的每个实例都会有一个 `tags` 属性，可以用来获取相关的 `TaggedItems` :

```
>>> b = Bookmark(url='https://www.djangoproject.com/')
>>> b.save()
>>> t1 = TaggedItem(content_object=b, tag='django')
>>> t1.save()
>>> t2 = TaggedItem(content_object=b, tag='python')
>>> t2.save()
>>> b.tags.all()
[<TaggedItem: django>, <TaggedItem: python>]
```

New in Django 1.7.

定义一个 `GenericRelation` 伴有 `related_query_name` 可以允许从相关联的对象中查询。

```
tags = GenericRelation(TaggedItem, related_query_name='bookmarks
')
```

这允许你从 `TaggedItem` 执行过滤筛选, 排序, 和其他的查询操作 `Bookmark` ：

```
>>> # Get all tags belonging to books containing `django` in the
 url
>>> TaggedItem.objects.filter(bookmarks__url__contains='django')
[<TaggedItem: django>, <TaggedItem: python>]
```

就像 `GenericForeignKey` 接受 content-type 和 object-ID 字段的命名为参数，`GenericRelation` 也是一样的。如果一个model的 generic foreignkey 字段使用的不是默认的命名,当你创建一个 `GenericRelation` 时一定要显示的传递这个字段的命名给它。例如，假如 `TaggedItem` model关联到上述所用的字段用 `content_type_fk` 和 `object_primary_key` 两个名称来创建一个 generic foreign key,然后一个 `GenericRelation` 需要这样定义:

```
tags = GenericRelation(TaggedItem,
                       content_type_field='content_type_fk',
                       object_id_field='object_primary_key')
```

当然，如果你没有添加一个反向的关系，你可以手动做相同类型的查找：

```
>>> b = Bookmark.objects.get(url='https://www.djangoproject.com/
')
>>> bookmark_type = ContentType.objects.get_for_model(b)
>>> TaggedItem.objects.filter(content_type__pk=bookmark_type.id,
...                           object_id=b.id)
[<TaggedItem: django>, <TaggedItem: python>]
```

注意，如果一个 `GenericRelation` model 在它的 `GenericForeignKey` 的 `ct_field` or `fk_field` 使用了非默认值，(for example, if you had a `Comment` model that uses `ct_field="object_pk"`), 你就要设置 `GenericRelation` 中的 `content_type_field` 和/或 `object_id_field` 来分别匹配 `GenericForeignKey` 中的 `ct_field` 和 `fk_field` :

```
comments = fields.GenericRelation(Comment, object_id_field="object_pk")
```

同时请注意,如果你删除了一个具有 `GenericRelation` 的对象, 任何以 `GenericForeignKey` 指向他的对象也会被删除. 在上面的例子中, 如果一个 `Bookmark` 对象被删除了,任何指向它的 `TaggedItem` 对象也会被同时删除。.

不同于 `ForeignKey` , `GenericForeignKey` 并不接受一个 `on_delete` 参数来控制它的行为：如果你非常渴望这种可控制行为，你应该不使用 `GenericRelation` 来避免这种级联删除，并且这种行为控制也可以通过 `pre_delete` 信号来提供。

通用关系和聚合

*Django’s database aggregation API*不能与 `GenericRelation` 配合使用。例如，您可能会试图尝试以下操作：

```
Bookmark.objects.aggregate(Count('tags'))
```

这将不会正确工作，然而generic relation添加了额外的查询集过滤来保证正确的内容类型, 但是 `aggregate()` 方法并没有被考虑进来。现在， 如果你需要在generic relations使用聚合，你只能不通过聚合API来计算他们。

Generic relation在表单中

`django.contrib.contenttypes.forms` 模块提供：

- BaseGenericInlineFormSet
- 用于 `GenericForeignKey` 的表单工厂，`generic_inlineformset_factory()` 。

class `BaseGenericInlineFormSet` [source]

自1.7版起已弃用：此类过去在 `django.contrib.contenttypes.generic` 中定义。将从Django 1.9中删除从此旧位置导入的支持。

`generic_inlineformset_factory` (*model*, *form=ModelForm*, *formset=BaseGenericInlineFormSet*, *ct_field="content_type"*, *fk_field="object_id"*, *fields=None*, *exclude=None*, *extra=3*, *can_order=False*, *can_delete=True*,

max_num=None, *formfield_callback=None*, *validate_max=False*, *for_concrete_model=True*, *min_num=None*, *validate_min=False*)[source]

使用 `modelformset_factory()` 返回 `GenericInlineFormSet` 。

如果它们分别与默认值， `content_type` 和 `object_id` 不同，则必须提供 `ct_field` 和 `fk_field` 。其他参数与 `modelformset_factory()` 和 `inlineformset_factory()` 中记录的参数类似。

`for_concrete_model` 参数对应于 `GenericForeignKey` 上的 `for_concrete_model` 参数。

自1.7版起已弃用：此函数用于在 `django.contrib.contenttypes.generic` 中定义。将从Django 1.9中删除从此旧位置导入的支持。

Changed in Django 1.7:

`min_num` 和 `validate_min` 。

管理中的通用关系

`django.contrib.contenttypes.admin` 模块提供 `GenericTabularInline` 和 `GenericStackedInline` （ `GenericInlineModel` 的子类别）

这些类和函数确保了generic relations在forms 和 admin的使用。有关详细信息，请参阅*model formset*和*admin*文档。

class `GenericInlineModelAdmin` [source]

`GenericInlineModelAdmin` 类继承了来自 `InlineModelAdmin` 类的所有属性。但是，它添加了一些自己的用于处理通用关系：

`ct_field`

模型上的 `ContentType` 外键字段的名称。默认为 `content_type` 。

`ct_fk_field`

表示相关对象的ID的整数字段的名称。默认为 `object_id` 。

自1.7版起已弃用：此类过去在 `django.contrib.contenttypes.generic` 中定义。将从Django 1.9中删除从此旧位置导入的支持。

class `GenericTabularInline` [source]

class `GenericStackedInline` [source]

`GenericInlineModelAdmin` 的子类，分别具有堆叠和表格布局。

自1.7版起已弃用：这些类以前在 `django.contrib.contenttypes.generic` 中定义。将从Django 1.9中删除从此旧位置导入的支持。

简单的页面应用程序

Django有一个可选的“简单页面”的应用它可以让你存储简单的扁平化结构的HTML内容在数据库中，你可以通过Django的管理界面和一个Python API处理要管理内容。

一个浮动页面是一个简单的包含有URL，标题和内容的对象。使用它作为一次性的，特殊用途的页面，比如“关于我们”或者“隐私政策”的页面，那些你想要保存在数据库，但是你又不想开发一个自定义的Django应用。

一个简单的页面应用程序可以使用自定义模板或者默认模板，系统的单页面模板。它可以和一个或多个站点相关联。

如果您希望将内容放在自定义模板中，该内容字段可能会任意的留下空白。

这里有一些基于Django的简单页面的示例：

- http://www.lawrence.com/about/contact/
- http://www2.ljworld.com/site/rules/

安装

按照下面这些步骤，安装单页面的应用

1. 通过向您的 `INSTALLED_APPS` 添加 `'django.contrib.sites'` ，安装 `sites framework`

   另外，请确保您已将 `SITE_ID` 正确设置为设置文件所代表的网站的ID。这通常是 `1` （即 `SITE_ID = 1` 重新使用sites框架来管理多个网站，它可以是不同网站的ID。

2. 将 `'django.contrib.flatpages'` 添加到您的 `INSTALLED_APPS` 设置。

然后：

1. 在URLconf中添加条目。例如：

```
urlpatterns = [
    url(r'^pages/', include('django.contrib.flatpages.urls')
),
]
```

要么：

1. 将 `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'`

添加到您的 `MIDDLEWARE_CLASSES` 设置。
2. 运行命令 `manage.py migrate` 。

怎么运行的

`manage.py migrate` 在数据库中创建两个表： `django_flatpage` 和 `django_flatpage_sites` 。 `django_flatpage` 是一个简单的查找表，只是将URL映射到标题和一堆文本内容。 `django_flatpage_sites` 将平面页面与网站相关联。

使用URLconf

有多种方法可以将平面网页纳入您的URLconf。您可以将特定路径专用于平面页面：

```
urlpatterns = [
    url(r'^pages/', include('django.contrib.flatpages.urls')),
]
```

您还可以将其设置为“总线”模式。在这种情况下，将模式放置在其他url模式的末尾非常重要：

```
from django.contrib.flatpages import views

# Your other patterns here
urlpatterns += [
    url(r'^(?P<url>.*/)$', views.flatpage),
]
```

警告

如果您将 `APPEND_SLASH` 设置为 `False` ，则必须删除catchall模式中的斜杠，否则不会匹配尾部斜杠。

另一个常见的设置是为一组有限的已知网页使用平面网页，并对网址进行硬编码，因此您可以使用 `url` 模板标记来引用它们：

```
from django.contrib.flatpages import views

urlpatterns += [
    url(r'^about-us/$', views.flatpage, {'url': '/about-us/'}, n
ame='about'),
    url(r'^license/$', views.flatpage, {'url': '/license/'}, nam
e='license'),
]
```

使用中间件

`FlatpageFallbackMiddleware`可以完成所有的工作。

class `FlatpageFallbackMiddleware` [source]

每次任何Django应用程序引发404错误，此中间件检查flatpages数据库的请求的URL作为最后手段。Specifically, it checks for a flatpage with the given URL with a site ID that corresponds to the `SITE_ID` setting.

如果找到匹配，则遵循此算法：

- 如果flatpage有一个自定义模板，它将加载该模板。否则，它会加载模板 `flatpages/default.html` 。
- 它传递那个模板一个单一的上下文变量， `flatpage` ，这是平面对象。它在呈现模板时使用 `RequestContext` 。

如果结果网址引用有效的平面网页，则中间件将仅添加尾部斜杠和重定向（通过查看 `APPEND_SLASH` 设置）。重定向是永久的（301状态码）。

如果它没有找到匹配，请求继续照常处理。

中间件仅针对404s激活 - 不是500秒或任何其他状态代码的响应。

Flatpages将不应用视图中间件

由于 `FlatpageFallbackMiddleware` 仅在网址解析失败并生成404后才应用，因此返回的响应将不应用任何*view middleware*方法。只有通过正常URL解析成功路由到视图的请求才应用视图中间件。

请注意， `MIDDLEWARE_CLASSES` 的顺序很重要。通常，您可以将 `FlatpageFallbackMiddleware` 放在列表的结尾。这意味着它将在处理响应时首先运行，并确保任何其他响应处理中间件看到真实的平面响应，而不是404。

有关中间件的更多信息，请阅读*middleware docs*。

确保您的404模板工作正常

请注意， `FlatpageFallbackMiddleware` 只会在另一个视图成功生成404响应时执行。如果另一个视图或中间件类尝试生成404，但最终会引发异常，则响应将变为HTTP 500（“内部服务器错误”），并且 `FlatpageFallbackMiddleware` 将不会尝试提供平面页。

如何添加，更改和删除flatpages

通过管理界面

如果您已激活自动Django管理界面，您应该在管理索引页上看到一个“Flatpages”部分。在编辑系统中的任何其他对象时编辑平铺页。

通过Python API

class `FlatPage` [source]

平铺页由标准的*Django model*表示，它位于django / contrib / flatpages / models.py中。您可以通过*Django database API*访问平面对象。

检查重复的平面网址。

如果您通过自己的代码添加或修改广告页，则可能需要检查同一网站中的重复的平展页网址。管理员使用的平面表单执行此验证检查，并且可以从 `django.contrib.flatpages.forms.FlatPageForm` 导入并在您自己的视图中使用。

平板模板

默认情况下，通过模板 `flatpages/default.html` 呈现平铺页，但您可以覆盖特定平面页：在管理中，标题为“高级选项”的折叠字段集（单击将其展开）包含用于指定模板名称的字段。如果您通过Python API创建平面网页，则可以简单地将模板名称设置为 `FlatPage` 对象上的字段 `template_name` 。

创建 `flatpages/default.html` 模板是您的责任；在模板目录中，只需创建一个包含文件 `default.html` 的 `flatpages` 目录。

平板模板传递单个上下文变量， `flatpage` ，这是平面对象。

以下是一个示例 `flatpages/default.html` 模板：

```
<!DOCTYPE html>
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content }}
</body>
</html>
```

由于您已将原始HTML输入到平面页面的管理页面，因此 `flatpage.title` 和 `flatpage.content` 都标记为而不是 *automatic HTML escaping*。

获取列表 `FlatPage`

flatpages应用提供了一个模板代码，可让您遍历*current site*上的所有可用平面网页。

Like all custom template tags, you'll need to *load its custom tag library* before you can use it. 加载库后，您可以通过 `get_flatpages` 标记检索所有当前的平铺页：

```
{% load flatpages %}
{% get_flatpages as flatpages %}
<ul>
    {% for page in flatpages %}
        <li><a href="{{ page.url }}">{{ page.title }}</a></li>
    {% endfor %}
</ul>
```

显示 `registration_required`

默认情况下， `get_flatpages` 模板标签只会显示标记为 `registration_required = False t3 >。` 如果要显示注册保护的纯页，则需要使用 `for` 子句指定已认证的用户。

例如：

```
{% get_flatpages for someuser as about_pages %}
```

如果您提供匿名用户，则 `get_flatpages` 的行为将与您未提供用户的行为相同，即只显示公开的页面。

根据基本网址限制平面广告

可以应用可选参数 `starts_with` ，以将返回的页面限制为以特定基本URL开头的页面。此参数可以作为字符串传递，或作为要从上下文解析的变量传递。

例如：

```
{% get_flatpages '/about/' as about_pages %}
{% get_flatpages about_prefix as about_pages %}
{% get_flatpages '/about/' for someuser as about_pages %}
```

整合 `django.contrib.sitemaps`

class `FlatPageSitemap` [source]

`sitemaps.` FlatPageSitemap类查看为当前 `SITE_ID` 定义的所有公开可见的 `flatpages` （请参阅 `sites documentation` 这些条目仅包含 `location` 属性 - 不是 `lastmod` ， `changefreq` 或 `priority` 。

Changed in Django 1.8:

此类可从旧版本的Django中的 `django.contrib.sitemaps.FlatPageSitemap` 获得。

示例

这里是一个使用 `FlatPageSitemap` 的URLconf示例：

```
from django.conf.urls import url
from django.contrib.flatpages.sitemaps import FlatPageSitemap
from django.contrib.sitemaps.views import sitemap

urlpatterns = [
    # ...

    # the sitemap
    url(r'^sitemap\.xml$', sitemap,
        {'sitemaps': {'flatpages': FlatPageSitemap}},
        name='django.contrib.sitemaps.views.sitemap'),
]
```

重定向应用

Django 原生自带一个可选的重定向应用。它将简单的重定向保存到数据库中并处理重定向。它默认使用HTTP 响应状态码 `301 Moved Permanently` 。

安装

请依照下面的步骤安装重定向应用：

1. 确保 `django.contrib.sites` 框架已经安装。
2. 添加' `django.contrib.redirects` '到 `INSTALLED_APPS` 设置中。
3. 添加' `django.contrib.redirects.middleware.RedirectFallbackMiddleware` '到 `MIDDLEWARE_CLASSES` 设置中。
4. 运行命令 `manage.py migrate` 。

它是如何工作的

`manage.py migrate` 在数据库中创建一张 `django_redirect` 表。它是一张简单的查询表，具有 `site_id` 、 `old_path` 和 `new_path` 字段。

`RedirectFallbackMiddleware` 完成所有的工作。每当Django 的应用引发一个404 错误，该中间件将到重定向数据库中检查请求的URL。它会根据 `old_path` 和 `SITE_ID` 设置的站点ID 查找重定向的路径。

- 如果找到匹配的记录且 `new_path` 不为空，它将使用301("Moved Permanently")重定向到 `new_path` 。你可以子类化 `RedirectFallbackMiddleware` 并设置 `response_redirect_class` 为 `django.http.HttpResponseRedirect` 来使用302 Moved Temporarily 重定向。
- 如果找到匹配的记录而 `new_path` 为空，它将发送一个410 ("Gone") HTTP 头和空（没有内容的）响应。
- 如果没有找到匹配的记录，请求将继续正常处理。

这个中间件只针对404 错误启用 —— 不能用于500 或其它状态码。

注意 `MIDDLEWARE_CLASSES` 的顺序很重要。通常可以将 `RedirectFallbackMiddleware` 放在列表的最后，因为它最后执行。

更多的信息可以阅读中间件的文档。

如何添加、修改和删除重定向

通过Admin 接口

如果你已经启用Django 自动生成的 `Admin` 接口，你应该可以在 `Admin` 的主页看到“Redirects”部分。编辑这些重定向，就像编辑系统中的其它对象一样。

通过Python API

`class models.Redirect`

重定向通过一个标准的Django 模型表示，位于 `django/contrib/redirects/models.py` 。你可以通过Django 的数据库API 访问重定向对象。

中间件

`class middleware.RedirectFallbackMiddleware`

你可以通过创建 `RedirectFallbackMiddleware` 的子类并覆盖 `response_gone_class` 和/或 `response_redirect_class` 来修改中间件使用的 `HttpResponse` 类。

`response_gone_class`

```
New in Django 1.7.
```

`HttpResponse` 类，用于找不到请求路径的 `Redirect` 或找到的 `new_path` 值为空的时候。

默认为 `HttpResponseGone` 。

`response_redirect_class`

```
New in Django 1.7.
```

处理重定向的 `HttpResponse` 类。

默认为 `HttpResponsePermanentRedirect` 。

> 译者：Django 文档协作翻译小组，原文：Redirects。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

信号

Django包含一个“信号的分发器”，允许解耦的应用在信号出现在框架的任何地方时，都能获得通知。简单来说，信号允许指定的 发送器通知一系列的接收器，一些操作已经发生了。当一些代码会相同事件感兴趣时，会十分有帮助。

Django 提供了一系列的内建信号，允许用户的代码获得DJango的特定操作的通知。这包含一些有用的通知：

- `django.db.models.signals.pre_save` & `django.db.models.signals.post_save`

  在模型 `save()` 方法调用之前或之后发送。

- `django.db.models.signals.pre_delete` & `django.db.models.signals.post_delete`

  在模型 `delete()` 方法或查询集的 `delete()` 方法调用之前或之后发送。

- `django.db.models.signals.m2m_changed`

  模型上的 `ManyToManyField` 修改时发送。

- `django.core.signals.request_started` & `django.core.signals.request_finished`

  Django建立或关闭HTTP 请求时发送。

关于完整列表以及每个信号的完整解释，请见内建信号的文档 。

你也可以定义和发送你自己的自定义信号；见下文。

监听信号

你需要注册一个接收器函数来接受信号，它在信号使用 `Signal.connect()` 发送时被调用：

`Signal.``connect` (*receiver*[, *sender=None*, *weak=True*, *dispatch_uid=None*])

Parameters:

- **receiver** – 和这个信号连接的回调函数。详见接收器函数。
- **sender** – 指定一个特定的发送器，来从它那里接受信号。详见连接由指定发送器发送的信号。
- **weak** – DJango通常以弱引用储存信号处理器。这就是说，如果你的接收器是个局部变量，可能会被垃圾回收。当你调用信号的 `connect()` 方法是，传递 `weak=False` 来防止这样做。
- **dispatch_uid** – 一个信号接收器的唯一标识符，以防信号多次发送。详见防止

重复的信号。

让我们来看一看它如何通过注册在每次在HTTP请求结束时调用的信号来工作。我们将会连接到 `request_finished` 信号。

接收器函数

首先，我们需要定义接收器函数。接受器可以是Python函数或者方法：

```
def my_callback(sender, **kwargs):
    print("Request finished!")
```

注意函数接受 `sender` 函数，以及通配符关键字参数(`**kwargs`)；所有信号处理器都必须接受这些参数。

我们过一会儿再关注发送器，现在先看一看 `**kwargs` 参数。所有信号都发送关键字参数，并且可以在任何时候修改这些关键字参数。在 `request_finished` 的情况下，它被记录为不发送任何参数，这意味着我们可能需要像 `my_callback(sender)` 一样编写我们自己的信号处理器。

这是错误的 -- 实际上，如果你这么做了，Django会抛出异常。这是因为无论什么时候信号中添加了参数，你的接收器都必须能够处理这些新的参数。

连接接收器函数

有两种方法可以将一个接收器连接到信号。你可以采用手动连接的方法：

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

或者使用 `receiver()` 装饰器来自动连接：

`receiver` (*signal*)

Parameters:

signal – A signal or a list of signals to connect a function to.

下面是使用装饰器连接的方法：

```
from django.core.signals import request_finished
from django.dispatch import receiver

@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

现在，我们的 `my_callback` 函数会在每次请求结束时调用。

这段代码应该放在哪里？

严格来说，信号处理和注册的代码应该放在你想要的任何地方，但是推荐避免放在应用的根模块和 `models` 模块中，以尽量减少产生导入代码的副作用。

实际上，信号处理通常定义在应用相关的 `signals` 子模块中。信号接收器在你应用配置类中的 `ready()` 方法中连接。如果你使用；额 `receiver()` 装饰器，只是在 `ready()` 内部导入 `signals` 子模块就可以了。

Changed in Django 1.7:

由于 `ready()` 并不在Django之前版本中存在，信号的注册通常在 `models` 模块中进行。

注意

`ready()` 方法会在测试期间执行多次，所以你可能想要防止重复的信号，尤其是打算在测试中发送它们的情况。

连接由指定发送器发送的信号

一些信号会发送多次，但是你只想接收这些信号的一个确定的子集。例如，考虑 `django.db.models.signals.pre_save` 信号，它在模型保存之前发送。大多数情况下，你并不需要知道 任何模型何时保存 -- 只需要知道一个特定的模型何时保存。

在这些情况下，你可以通过注册来接收只由特定发送器发出的信号。对于 `django.db.models.signals.pre_save` 的情况， 发送者是被保存的模型类，所以你可以认为你只需要由某些模型发出的信号：

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel

@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs):
    ...
```

`my_handler` 函数只在 `MyModel` 实例保存时被调用。

不同的信号使用不同的对象作为他们的发送器；对于每个特定信号的细节，你需要查看内建信号的文档。

防止重复的信号

在一些情况下，向接收者发送信号的代码可能会执行多次。这会使你的接收器函数被注册多次，并且导致它对于同一信号事件被调用多次。

如果这样的行为会导致问题（例如在任何时候模型保存时使用信号来发送邮件），传递一个唯一的标识符作为 `dispatch_uid` 参数来标识你的接收器函数。标识符通常是一个字符串，虽然任何可计算哈希的对象都可以。最后的结果是，对于每个唯一的 `dispatch_uid` 值，你的接收器函数都只被信号调用一次：

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_id
entifier")
```

定义和发送信号

你的应用可以利用信号功能来提供自己的信号。

定义信号

class `Signal` ([*providing_args=list*])

所有信号都是 `django.dispatch.Signal` 的实例。 `providing_args` 是一个列表，包含参数的名字，它们由信号提供给监听者。理论上是这样，但是实际上并没有任何检查来保证向监听者提供了这些参数。

例如：

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings",
"size"])
```

这段代码声明了 `pizza_done` 信号，它向接受者提供 `toppings` 和 `size` 参数。

要记住你可以在任何时候修改参数的列表，所以首次尝试的时候不需要完全确定API。

发送信号

Django中有两种方法用于发送信号。

`Signal.``send` (*sender*, ***kwargs*)

`Signal.``send_robust` (*sender*, ***kwargs*)

调用 `Signal.send()` 或者 `Signal.send_robust()` 来发送信号。你必须提供 `sender` 参数（大多数情况下它是一个类），并且可以提供尽可能多的关键字参数。

例如，这样来发送我们的 `pizza_done` 信号：

```
class PizzaStore(object):
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings
, size=size)
        ...
```

`send()` 和 `send_robust()` 都会返回一个含有二元组的列表 `[(receiver, response), ... ]`，它代表了被调用的接收器函数和他们的响应值。

`send()` 与 `send_robust()` 在处理接收器函数产生的异常时有所不同。 `send()` 不会 捕获任何由接收器产生的异常。它会简单地让错误往上传递。所以在错误产生的情况，不是所有接收器都会获得通知。

`send_robust()` 捕获所有继承自Python `Exception` 类的异常，并且确保所有接收器都能得到信号的通知。如果发生了错误，错误的实例会在产生错误的接收器的二元组中返回。

New in Django 1.8:

调用 `send_robust()` 的时候，所返回的错误的 `__traceback__` 属性上会带有 traceback。

断开信号

`Signal.``disconnect` ([*receiver=None*, *sender=None*, *weak=True*, *dispatch_uid=None*])

调用 `Signal.disconnect()` 来断开信号的接收器。 `Signal.connect()` 中描述了所有参数。如果接收器成功断开，返回 `True` ，否则返回 `False` 。

`receiver` 参数表示要断开的已注册接收器。如果 `dispatch_uid` 用于定义接收器，可以为 `None` 。

Changed in Django 1.8:

增加了返回的布尔值。

> 译者：Django 文档协作翻译小组，原文：Signals。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

系统检查框架

New in Django 1.7.

系统检查框架是为了验证Django项目的一系列静态检查。它可以检测到普遍的问题，并且提供如何修复的提示。这个框架可以被扩展，所以你可以轻易地添加你自己的检查。

检查可以由 `check` 命令显式触发。检查会在大多数命令之前隐式触发，包括 `runserver` 和 `migrate` 。由于性能因素，检查不作为在部署中使用的WSGI栈的一部分运行。如果你需要在你的部署服务器上运行系统检查，显式使用 `check` 来触发它们。

严重的错误会完全阻止Django命令(像 `runserver`)的运行。少数问题会通过控制台来报告。如果你检查了警告的原因，并且愿意无视它，你可以使用你项目设置文件中的 `SILENCED_SYSTEM_CHECKS` 设置，来隐藏特定的警告。

系统检查参考中列出了所有Django可执行的所有检查。

编写你自己的检查

这个框架十分灵活，允许你编写函数，执行任何其他类型的所需检查。下面是一个桩（stub）检查函数的例子：

```
from django.core.checks import register

@register()
def example_check(app_configs, **kwargs):
    errors = []
    # ... your check logic here
    return errors
```

检查函数必须接受 `app_configs` 参数；这个参数是要被检查的应用列表。如果是None，检查会运行在项目中所有安装的应用上。 `**kwargs` 参数用于进一步的扩展。

消息

这个函数必须返回消息的列表。如果检查的结果中没有发现问题，检查函数必须返回一个空列表。

class `CheckMessage` (*level*, *msg*, *hint*, *obj=None*, *id=None*)

由检查方法产生的警告和错误必须是 `CheckMessage` 的示例。 `CheckMessage` 的实例封装了一个可报告的错误或者警告。它同时也提供了可应用到消息的上下文或者提示，以及一个用于过滤的唯一的标识符。

它的概念非常类似于消息框架或者 日志框架中的消息。消息使用表明其严重性的 `level` 来标记。

构造器的参数是：

`level`

The severity of the message. Use one of the predefined values: `DEBUG` , `INFO` , `WARNING` , `ERROR` , `CRITICAL` . If the level is greater or equal to `ERROR` , then Django will prevent management commands from executing. Messages with level lower than `ERROR` (i.e. warnings) are reported to the console, but can be silenced.

`msg`

A short (less than 80 characters) string describing the problem. The string should *not* contain newlines.

`hint`

A single-line string providing a hint for fixing the problem. If no hint can be provided, or the hint is self-evident from the error message, the hint can be omitted, or a value of `None` can be used.

`obj`

Optional. An object providing context for the message (for example, the model where the problem was discovered). The object should be a model, field, or manager or any other object that defines `__str__` method (on Python 2 you need to define `__unicode__` method). The method is used while reporting all messages and its result precedes the message.

`id`

Optional string. A unique identifier for the issue. Identifiers should follow the pattern `applabel.X001` , where `X` is one of the letters `CEWID` , indicating the message severity (`C` for criticals, `E` for errors and so). The number can be allocated by the application, but should be unique within that application.

也有一些快捷方式，使得创建通用级别的消息变得简单。当使用这些方法时你可以忽略 `level` 参数，因为它由类名称暗示。

class `Debug` (*msg*, *hint*, *obj=None*, *id=None*)

class `Info` (*msg*, *hint*, *obj=None*, *id=None*)

class `Warning` (*msg*, *hint*, *obj=None*, *id=None*)

class `Error` (*msg*, *hint*, *obj=None*, *id=None*)

class `Critical` (*msg*, *hint*, *obj=None*, *id=None*)

消息是可比较的。你可以轻易地编写测试：

```
from django.core.checks import Error
errors = checked_object.check()
expected_errors = [
    Error(
        'an error',
        hint=None,
        obj=checked_object,
        id='myapp.E001',
    )
]
self.assertEqual(errors, expected_errors)
```

注册和标记检查

最后，你的检查函数必须使用系统检查登记处来显式注册。

`register` (**tags)(function*)

你可以向 `register` 传递任意数量的标签来标记你的检查。Tagging checks is useful since it allows you to run only a certain group of checks. For example, to register a compatibility check, you would make the following call:

```
from django.core.checks import register, Tags

@register(Tags.compatibility)
def my_check(app_configs, **kwargs):
    # ... perform compatibility checks and collect errors
    return errors
```

New in Django 1.8.

你可以注册“部署的检查”，它们只和产品配置文件相关，像这样：

```
@register(Tags.security, deploy=True)
def my_check(app_configs, **kwargs):
    ...
```

这些检查只在 `--deploy` 选项传递给 `check` 命令的情况下运行。

你也可以通过向 `register` 传递一个可调用对象（通常是个函数）作为第一个函数，将 `register` 作为函数使用，而不是一个装饰器。

下面的代码和上面等价：

```
def my_check(app_configs, **kwargs):
    ...
register(my_check, Tags.security, deploy=True)
```

Changed in Django 1.8:

添加了将注册用作函数的功能。

字段、模型和管理器检查

在一些情况下，你并不需要注册检查函数 -- 你可以直接使用现有的注册。

字段、方法和模型管理器都实现了 `check()` 方法，它已经使用检查框架注册。如果你想要添加额外的检查，你可以扩展基类中的实现，进行任何你需要的额外检查，并且将任何消息附加到基类生成的消息中。强烈推荐你将每个检查分配到单独的方法中。

考虑一个例子，其中你要实现一个叫做 `RangedIntegerField` 的自定义字段。这个字段向 `IntegerField` 的构造器中添加 `min` 和 `max` 参数。你可能想添加一个检查，来确保用户提供了小于等于最大值的最小值。下面的代码段展示了如何实现这个检查：

```
from django.core import checks
from django.db import models

class RangedIntegerField(models.IntegerField):
    def __init__(self, min=None, max=None, **kwargs):
        super(RangedIntegerField, self).__init__(**kwargs)
        self.min = min
        self.max = max

    def check(self, **kwargs):
        # Call the superclass
        errors = super(RangedIntegerField, self).check(**kwargs)

        # Do some custom checks and add messages to `errors`:
        errors.extend(self._check_min_max_values(**kwargs))

        # Return all errors and warnings
        return errors

    def _check_min_max_values(self, **kwargs):
        if (self.min is not None and
                self.max is not None and
                self.min > self.max):
            return [
                checks.Error(
                    'min greater than max.',
                    hint='Decrease min or increase max.',
                    obj=self,
                    id='myapp.E001',
                )
            ]
        # When no error, return an empty list
        return []
```

如果你想要向模型管理器添加检查，应该在你的 `Manager` 的子类上执行同样的方法。

如果你想要向模型类添加检查，方法也大致相同：唯一的不同是检查是类方法，并不是实例方法：

```
class MyModel(models.Model):
    @classmethod
    def check(cls, **kwargs):
        errors = super(MyModel, cls).check(**kwargs)
        # ... your own checks ...
        return errors
```

> 译者：Django 文档协作翻译小组，原文：System check framework。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

The “sites” framework

Django 原生带有一个可选的“sites”框架。它是一个钩子，用于将对象和功能与特定的站点关联，它同时还是域名和你的Django 站点名称之间的对应关系所保存的地方。

如果你的Django 不只为一个站点提供支持，而且你需要区分这些不同的站点，你就可以使用它。

Sites 框架主要依据一个简单的模型：

class `models.``Site`

用来存储Web站点的 `domain` 和 `name` 属性的模型

`domain`

与Web站点关联的域名。

`name`

Web 站点的名称。

`SITE_ID` 设置指定与特定的设置文件关联的 `Site` 对象在数据库中ID。如果省略该设置， `get_current_site()` 函数将会通过比较 `domain` 与 `request.get_host()` 方法中得到的主机名，来得到当前的Site。

怎样使用取决于你，但是django自动的在几个方面通过一些简单的约定使用它。

示例

为什么要使用Sites 框架？这点通过实例来理解的效果最好

关联内容到多个站点

通过Django开发的站点LJWorld.com 和Lawrence.com 是位于Lawrence, Kansas 的同一家机构Lawrence Journal-World newspaper 运营的。LJWorld.com 关注新闻，而Lawrence.com 关注当地的环境问题。但是有时编辑需要发布同一篇文章到两个站点。

无脑的解决方法是要求站点发布者发布同一内容两次：一次到LJWorld.com，一次到 Lawrence.com。但这是很低效的行为，而且在数据库中必须存储同一内容很多次（多副本存储，浪费资源）。

最好的解决方法很简单：两个站点用相同的文章数据库，一篇文章可以关联一个或者多个站点。用Django 模型的术语，它通过 `Article` 模型的一个 多对多字段 表示：

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    sites = models.ManyToManyField(Site)
```

这很快很好的完成了几件事：

- 它使得站点编辑者利用一个接口(Django admin)编辑多站点上的所有内容。
- 它意味着同一个内容不用往数据库存入两次；在数据库中仅仅只有一条记录。
- 对于两个站点，开发者可以使用相同的Django 视图代码。显示内容的视图代码需要检查，以确保请求的内容属于当前的站点。就像下面一样:

  ```
  from django.contrib.sites.shortcuts import get_current_site

  def article_detail(request, article_id):
      try:
          a = Article.objects.get(id=article_id, sites__id=get_current_site(request).id)
      except Article.DoesNotExist:
          raise Http404("Article does not exist on this site")
      # ...
  ```

关联内容到单独的站点

类似地，你可以用 `ForeignKey` 关联一个模型到 `Site` 模型实现多对一关系。

例如，一篇文章只允许在一个单独的站点，你应该像这样用模型：

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    site = models.ForeignKey(Site)
```

这个好处和上节描述的好处是相同的。

在视图中获得当前的Site

你可以在Django 视图中使用Sites 框架基于正在调用的视图所在的Site 实现特定的功能。例如：

```
from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
        pass
    else:
        # Do something else.
        pass
```

当然，这样硬编码Site ID 比较丑陋。这种硬编码是你最需要尽快修复的。完成这件事情的更清洁的方法是检查当前站点的域名：

```
from django.contrib.sites.shortcuts import get_current_site

def my_view(request):
    current_site = get_current_site(request)
    if current_site.domain == 'foo.com':
        # Do something
        pass
    else:
        # Do something else.
        pass
```

它还有一个优点是检查Sites 框架是否安装，如果没有安装将返回一个 `RequestSite` 实例。

如果你不能访问request 对象，你可以使用 `Site` 模型管理器的 `get_current()` 方法。此时，你需要确保你的设置文件包含 `SITE_ID` 设置。下面的示例与前面的示例等同：

```
from django.contrib.sites.models import Site

def my_function_without_request():
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
        pass
    else:
        # Do something else.
        pass
```

显示当前的域名

LJWorld.com 和Lawrence.com 都具有邮件通知功能，它让读者注册以在新闻发生时获得通知。这很简单：读者通过网页表单注册，然后立即收到一封邮件说“感谢您的订阅”。

将这个注册过程的代码实现两次是低效而冗余的，所以这两个站点在后台使用相同的代码。但是每个Site 的"感谢您的订阅"的通知需要不同。通过使用 `Site` 对象，我们可以抽象这个通知并利用当前Site 的 `name` 和 `domain` 的值。

下面是该表单处理视图的一个例子：

```
from django.contrib.sites.shortcuts import get_current_site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    current_site = get_current_site(request)
    send_mail('Thanks for subscribing to %s alerts' % current_si
te.name,
        'Thanks for your subscription. We appreciate it.\n\n-The
 %s team.' % current_site.name,
        'editor@%s' % current_site.domain,
        [user.email])

    # ...
```

在Lawrence.com 网站上，这封邮件的标题为"Thanks for subscribing to lawrence.com alerts."。在LJWorld.com 网站上，这封邮件的标题为"Thanks for subscribing to LJWorld.com alerts."。邮件体的行为相同。

注意，更加灵活（但是更沉重）的方法是使用Django 的模板系统。假设Lawrence.com 和LJWorld.com 具有不同的模板目录（ `DIRS` ），你可以很容易地根据模板系统写出：

```
from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    subject = loader.get_template('alerts/subject.txt').render(C
ontext({}))
    message = loader.get_template('alerts/message.txt').render(C
ontext({}))
    send_mail(subject, message, 'editor@ljworld.com', [user.emai
l])

    # ...
```

在这种情况下，你必须为LJWorld.com 和Lawrence.com 模板目录都创建 `subject.txt` 和 `message.txt` 模板文件。它更灵活，但是也更复杂。

尽可能地发掘 `Site` 对象的用法以删除不需要的复杂性和冗余是个不错的主意。

获取当前域名的url全路径

Django 的 `get_absolute_url()` 可以很方便地获得对象不带域名的URL，但是某些情况下，你可能想显示完整的URL，带有 `http://` 和域名以及其它部分。要实现这点，你可以使用Sites 框架。一个简单的示例：

```
>>> from django.contrib.sites.models import Site
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://example.com/mymodel/objects/3/'
```

启用Sites 框架

按照以下步骤启用Sites 框架：

1. 添加 `'django.contrib.sites'` 到你的 `INSTALLED_APPS` 设置中。
2. 定义 `SITE_ID` 设置：

    ```
    SITE_ID = 1
    ```

3. 运行 `migrate` 。

`django.contrib.sites` 注册一个 `post_migrate` 信号处理器，它创建一个默认的Site `example.com` ，其域名为 `example.com` 。在Django 创建测试数据库之后，也会创建该Site。你可以使用数据迁移来为你的项目设置正确的name 和domain。

为了在线上环境中启用多个Site，你应该为每个 `SITE_ID` 创建一个单独的设置文件（可以从一个共同的设置文件导入，以避免重复共享的配置），然后为每个Site指定合适的 `DJANGO_SETTINGS_MODULE` 。

缓存当前 `Site`

因为当前站点储存在数据库,每一次调用 `Site.objects.get_current()` 都会导致数据库查询。但是Django还是比这个聪明滴, 当前站点被放在缓存当中了, 所以后续的调用返回的都是缓存的数据而不是直接查询数据库。

如果出于一些原因你想要强制用数据库查询, 你可以告诉Django清除缓存，用下面这个方法 `Site.objects.clear_cache()` :

```
# First call; current site fetched from database.
current_site = Site.objects.get_current()
# ...

# Second call; current site fetched from cache.
current_site = Site.objects.get_current()
# ...

# Force a database query for the third call.
Site.objects.clear_cache()
current_site = Site.objects.get_current()
```

的 `CurrentSiteManager`

class `managers.``CurrentSiteManager`

如果 `Site` 在你的应用中非常的关键， 你可以考虑用 `CurrentSiteManager` 在你的模型中(s). 它是一个 model*manager*用来自动过滤，留下只与当前站点有关的数据查询 `Site` .

必须 `SITE_ID`

`CurrentSiteManager` 只有在你定义了 `SITE_ID` 在setting 中才起作用。

使用 `CurrentSiteManager` ，你只要直接把他添加到你的model 中。例如：

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

通过这个model, `Photo.objects.all()` 将会返回所有在数据库中的 `Photo` 对象，但是 `Photo.on_site.all()` 只会返回 与当前site相关的 `Photo` 对象, 这是根据 `SITE_ID` 在setting的设置。

换句话说，这两种表达方式是等价的:

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

`CurrentSiteManager` 是如何知道哪个 `Photo` 字段是 `Site` 的? 通常来说, `CurrentSiteManager` 查找一个 `ForeignKey` 它的名字叫 `site` 或者是一个 `ManyToManyField` 字段 ,叫做 `sites` 来筛选出. 如果你用名字不叫 `site` or `sites` 的字段来表示一个与 `Site` 对象相关联,,那么你就需要在你的model中显示得传递自定义的字段名给 `CurrentSiteManager` 。下面的model, 它有一个字段叫做 `publish_on` , 说明了这个问题 :

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

如果你尝试使用 `CurrentSiteManager` 并且传递了一个并不存在的字段名称给他, Django 就会引发一个 `ValueError` .

最后, 注意你可能会想要保持一个正常的 (non-site-specific) `Manager` 在你的model, 虽然你使用了 `CurrentSiteManager` . 就像 *manager documentation* 当中的解释那样,如果你手动定义了一个manager,Django是不会为你自动创建 `objects = models.` Manager() manager。也请注意某些 Django组件 –即, Django admin site 和通用视图– 使用的是 *first* 定义 在你model中的manager,所以如果你希望你的admin site可以连接到所有对象 (不仅仅是特定的站点对象), 那就设置 `objects = models.` Manager() 在你的 model中, 并且在你定义 `CurrentSiteManager` 之前。

网站中间件

New in Django 1.7.

如果你经常使用这个模式 :

```
from django.contrib.sites.models import Site

def my_view(request):
    site = Site.objects.get_current()
    ...
```

这里有些方法可以防止这种重复调用。添加 `django.contrib.sites.middleware.CurrentSiteMiddleware` 到 `MIDDLEWARE_CLASSES` . 中间件设置 `site` 属性给每一次request对象, 所以你可以用 `request.site` 来获取当前site。

Django是如何使用的站点框架

虽然不强制要求你的网站使用site框架，但是我们鼓励你使用它，因为在一些地方Django利用它。 即使你的Django只在支持单个站点, 你也应该花两秒时间来给你的站点对象创建 `domain` 和 `name` ,并且设置它的ID在你的 `SITE_ID` setting中。

下面是Django 如何使用sites framework:

- 在 `redirects framework` ,每一个redirect都和特定的站点相关联。当Django查找一个 redirect, 它就考虑在当前的站点中查找。
- 在 `flatpages 框架` , 每一个flatpage 都被关联到特定的站点。当一个flatpage 被创建， 你指定它的 `Site` ,并且 `FlatpageFallbackMiddleware` 在返回flatpages 中检查当前站以显示。
- 在 `syndication framework` 中, 模板的 `title` and `description` 自动访问变量 `， 这个 [ Site ](#django.contrib.sites.models.Site "django.cont Site ](#django.contrib.sites.models.Site "django.contrib.sites.mod domain`，如果你不指定一个完全合格的域名。
- 在 `authentication framework` 中， `django.contrib.auth.views.login()` 视图传递当前 `Site` 名称的模板``.
- 快捷视图 (`django.contrib.contenttypes.views.shortcut`) 使用当前 `Site` 对象的的域 计算对象的URL。
- 在管理框架, "view on site" 链接使用当前 `Site` 算出将重定向的域名.

RequestSite objects

一些 *django.contrib*应用有利用到 sites framework 但是它们的架构不会*require* sites framework必须安装在你的数据库中。有些人不想, 或者不能安装site framework所要求的*able*在他们的数据库中。) 出于这种情况，framework 提供了一个 `django.contrib.sites.requests.RequestSite` 类，当你数据支持的站点框架不可用的时候做一个回退

class `requests.``RequestSite`

一个共享 `Site` （即，它具有 `domain` 和 `name` 属性）的主接口，但从Django `HttpRequest` 对象，而不是从数据库。

`__init__` (*request*)

将 `name` 和 `domain` 属性设置为 `get_host()` 的值。

自1.7版起已弃用：此类过去在 `django.contrib.sites.models` 中定义。旧的导入位置将工作，直到Django 1.9。

除了其 `__init__()` 方法采用 `HttpRequest` 对象， `RequestSite` 对象具有与正常 `Site` 它可以通过查看请求的域来推断 `domain` 和 `name` 。它具有 `save()` 和 `delete()` 方法来匹配 `Site` 的接口，但是方法产生 `NotImplementedError` 。

get_current_site shortcut

最后,为了避免重复的回退代码，site framework 提供了一个 `django.contrib.sites.shortcuts.get_current_site()` 功能。

`shortcuts.``get_current_site` (*request*)

这是函数是用来检查 `django.contrib.sites` 是否安装并且返回一个基于request的 `Site` 对象或者一个 `RequestSite` 对象。

自1.7版起已弃用：此函数用于在 `django.contrib.sites.models` 中定义。旧的导入位置将工作，直到Django 1.9。

Changed in Django 1.8:

如果未定义 `SITE_ID` 设置，此函数现在将根据 `request.get_host()` 查找当前站点。

Unicode数据

Django所有地方都原生地支持Unicode数据。只要你的数据库能存储数据，你就可以安全地把Unicode字符串传递到模板、模型和数据库中。

本文档告诉你如果当你写用到非ASCII的数据或者模板的应用时，你需要知道什么。

创建数据库

确认你的数据库配置可以存储任意字符串数据。一般来讲，这意味着给它一个UTF-8或者UTF-16的编码方式。如果你用了更具约束性的编码 – 例如latin1 (iso8859-1) – 你将无法存储某些特定的字符到数据库，并且这些信息也会丢失。

- MySQL用户，有关如何设置或更改数据库字符集编码的详细信息，请参阅MySQL手册。
- PostgreSQL用户，有关使用正确的编码创建数据库的详细信息，请参阅PostgreSQL手册（PostgreSQL 9中的第22.3.2节）。
- SQLite用户，没有什么你需要做的。SQLite总是使用UTF-8进行内部编码。

所有Django的数据库后端都自动将Unicode字符串转换为与数据库通信的适当编码。它们还自动将从数据库检索的字符串转换为Python Unicode字符串。你甚至不需要告诉Django你的数据库使用什么编码：这是透明地处理。

有关更多信息，请参阅下面的“数据库API”部分。

一般字符串处理

每当你使用Django字符串 - 例如，在数据库查找，模板渲染或其他地方 - 你有两个选择来编码这些字符串。您可以使用Unicode字符串，也可以使用使用UTF-8编码的普通字符串（有时称为“bytestrings”）。

在Python 3中，逻辑是相反的，这是正常的字符串是Unicode，当你想要专门创建一个bytestring，你必须在字符串前面加一个'b'。正如我们在1.5版本的Django代码中所做的，我们建议您从代码中的**future**库中导入 `unicode_literals` 。然后，当您特别要创建一个bytestring字面量时，在字符串前面加上'b'。

Python 2的遗产：

```
my_string = "This is a bytestring"
my_unicode = u"This is an Unicode string"
```

Python 2与unicode文字或Python 3：

```
from __future__ import unicode_literals

my_string = b"This is a bytestring"
my_unicode = "This is an Unicode string"
```

另请参见*Python 3 compatibility*。

警告：

一个字节不包含关于它的编码的任何信息。为此，我们必须做一个假设，Django假设所有的bytestrings都是UTF-8。

如果你传递一个字符串到Django已经编码的其他格式，事情会出错的有趣的方式。通常，Django在某个时刻会引发 `UnicodeDecodeError` 。

如果您的代码只使用ASCII数据，那么可以安全地使用正常的字符串，随意传递它们，因为ASCII是UTF-8的子集。

不要被欺骗，认为如果您的 `DEFAULT_CHARSET` 设置设置为 `'utf-8'` 以外的其他设置，您可以在您的bytestrings中使用其他编码！ `DEFAULT_CHARSET` 仅适用于作为模板呈现（和电子邮件）的结果生成的字符串。Django将始终假定内部bytestrings的UTF-8编码。原因是 `DEFAULT_CHARSET` 设置实际上不在您的控制之下（如果您是应用程序开发人员）。它是在安装和使用您的应用程序的人的控制下 - 如果该人选择不同的设置，您的代码必须仍然继续工作。Ergo，它不能依赖那个设置。

在大多数情况下，当Django处理字符串时，它会将它们转换为Unicode字符串，然后再做其他事情。所以，作为一般规则，如果你传递一个字节，准备在结果中接收一个Unicode字符串。

翻译字符串

除了Unicode字符串和bytestrings，有使用Django时可能遇到的第三种类型的字符串对象。框架的国际化特性引入了一个“延迟翻译”的概念 - 一个字符串，标记为已翻译，但实际的翻译结果直到对象在字符串中使用才被确定。在使用字符串之前翻译语言环境未知的情况下，此功能非常有用，即使字符串最初可能是在首次导入代码时创建的。

通常，你不必担心延迟翻译。只要注意，如果你检查一个对象，它声称是一个 `django.utils.functional.__proxy__` 对象，它是一个延迟的翻译。以延迟转换作为参数调用 `unicode()` 将在当前语言环境中生成一个Unicode字符串。

有关延迟翻译对象的更多详细信息，请参阅*internationalization*文档。

有用的实用功能

因为一些字符串操作一次又一次地出现，Django附带了一些有用的函数，应该使用Unicode和bytestring对象更容易一些。

转换函数

`django.utils.encoding` 模块包含一些方便在Unicode和bytestrings之间来回转换的函数。

- `smart_text(s, encoding='utf-8', strings_only=False, errors='stri` 将输入内容转换为Unicode编码， `encoding` 参数用来指定输入的编码.（例如，Django在处理表单输入数据时使用此内部数据，这可能不是UTF-8编码的。） `strings_only` 参数, 当设定为True, 输入的 numbers, booleans , `None` 并不会转换成字符类型 (保留原来的类型). `errors` 参数使用Python的 `unicode()` 函数接受的任何值来处理它的错误。

  如果传递 `smart_text()` 对象具有 `__unicode__` 方法，它将使用该方法进行转换。

- `force_text（s， encoding ='utf-8'， strings_only = False， errors` 与 `smart_text()` 完全相同。区别在于第一个参数是*lazy translation*实例。当 `smart_text()` 保留延迟翻译时， `force_text()` 将这些对象强制为Unicode字符串（导致翻译发生）。通常，您需要使用 `smart_text()` 。但是， `force_text()` 在绝对必须具有要处理的字符串的模板标记和过滤器中很有用，而不仅仅是可以转换为字符串的内容。

- `smart_bytes（s， encoding ='utf-8'， strings_only = False， error` 本质上与 `smart_text()` 相反。它强制第一个参数为一个字节。 `strings_only` 参数与 `smart_text()` 和 `force_text()` 具有相同的行为。这与Python的内置 `str()` 函数略有不同，但是在Django内部的几个地方需要区别。

通常，您只需要使用 `smart_text()` 。尽可能早地调用任何可能是Unicode或bytestring的输入数据，从那时起，可以将结果视为一直是Unicode。

URI和IRI处理

Web框架必须处理URL（这是一种IRI）。URL的一个要求是它们仅使用ASCII字符编码。但是，在国际环境中，您可能需要从IRI（非常宽松地说，URI）构造一个可以包含Unicode字符的URL。引用和转换IRI到URI可能有点棘手，因此Django提供了一些帮助。

- 函数 `django.utils.encoding.iri_to_uri()` 根据规范（ **RFC 3987**）的要求实现从IRI到URI的转换。
- The functions `django.utils.http.urlquote()` and `django.utils.http.urlquote_plus()` are versions of Python's standard `urllib.quote()` and `urllib.quote_plus()` that work with non-ASCII characters. （数据在编码前转换为UTF-8.）

这两组功能的目的略有不同，重要的是保持它们的直线。通常，您可以在IRI或URI路径的各个部分使用 `urlquote()` ，以便正确编码任何保留字符，例如'&'或'%'。然后，将 `iri_to_uri()` 应用于完整的IRI，并将任何非ASCII字符转换为正确的编

码值。

注意

从技术上讲，`iri_to_uri()` 在IRI规范中实现了完整的算法是不正确的。它没有（还）执行算法的国际域名编码部分。

`iri_to_uri()` 函数不会更改URL中允许的ASCII字符。因此，例如，当传递给 `iri_to_uri()` 时，字符'%'不会进一步编码。这意味着你可以传递一个完整的URL到这个函数，它不会弄乱查询字符串或类似的东西。

一个例子可能会在这里澄清一下：

```
>>> urlquote('Paris & Orléans')
'Paris%20%26%20Orl%C3%A9ans'
>>> iri_to_uri('/favorites/François/%s' % urlquote('Paris & Orlé
ans'))
'/favorites/Fran%C3%A7ois/Paris%20%26%20Orl%C3%A9ans'
```

如果仔细查看，可以看到第二个示例中由 `urlquote()` 生成的部分在传递到 `iri_to_uri()` 时没有双引号。这是一个非常重要和有用的功能。这意味着你可以构造你的IRI，而不必担心它是否包含非ASCII字符，然后，在结束，调用 `iri_to_uri()` 在结果。

类似地，Django提供 `django.utils.encoding.uri_to_iri()` ，它实现了根据 **RFC 3987**从URI到IRI的转换。它对除了不表示有效UTF-8序列的那些编码之外的所有百分比编码进行解码。

示例：

```
>>> uri_to_iri('/%E2%99%A5%E2%99%A5/?utf8=%E2%9C%93')
'/♥♥/?utf8=✓'
>>> uri_to_iri('%A9helloworld')
'%A9helloworld'
```

在第一个示例中，UTF-8字符和保留字符未引用。在第二个，百分比编码保持不变，因为它位于有效的UTF-8范围之外。

`iri_to_uri()` 和 `uri_to_iri()` 函数是幂等的，这意味着以下内容总是为真：

```
iri_to_uri(iri_to_uri(some_string)) == iri_to_uri(some_string)
uri_to_iri(uri_to_iri(some_string)) == uri_to_iri(some_string)
```

因此，您可以安全地在同一URI / IRI上多次调用它，而不会冒双重引用问题。

楷模

因为所有字符串都作为Unicode字符串从数据库返回，所以当Django从数据库中检索数据时，基于字符（CharField，TextField，URLField等）的模型字段将包含Unicode值。这是始终的情况，即使数据可以适合ASCII测试。

您可以在创建模型或填充字段时传递bytestrings，Django会在需要时将其转换为Unicode。

选择之间 `__str__()`

注意

如果你使用Python 3，可以跳过这一部分，因为你总是创建 `__str__()` 而不是 `__unicode__()` 。如果你想与Python 2兼容，你可以用 `python_2_unicode_compatible()` 装饰你的模型类。

缺省情况下使用Unicode的一个结果是，在打印模型中的数据时需要小心。

特别是，我们建议您实现 `__unicode__()` 方法，而不是为您的模型提供 `__str__()` 在 `__unicode__()` 方法中，您可以非常安全地返回所有字段的值，而不必担心它们是否适合一个字节。（Python的工作方式， `__str__()` 的结果是总是一个bytestring，即使你不小心尝试返回一个Unicode对象）。

你仍然可以在模型上创建一个 `__str__()` 方法，当然，除非你有很好的理由，否则你不应该这样做。Django的 `Model` 基类自动提供了调用 `__unicode__()` 的 `__str__()` 实现，并将结果编码为UTF-8。这意味着你通常只需要实现一个 `__unicode__()` 方法，并让Django在需要时处理强制的字节。

照顾 `get_absolute_url()`

网址只能包含ASCII字符。如果要从可能是非ASCII的数据片段构造URL，请小心地以适合URL的方式对结果进行编码。 `reverse()` 函数会自动处理此事件。

如果您手动构建网址（即而不是使用 `reverse()` 函数），则需要自己处理编码。在这种情况下，请使用上面中记录的 `iri_to_uri()` 和 `urlquote()` 函数。例如：

```
from django.utils.encoding import iri_to_uri
from django.utils.http import urlquote

def get_absolute_url(self):
    url = '/person/%s/?x=0&y=0' % urlquote(self.location)
    return iri_to_uri(url)
```

即使 `self.location` 类似于“Jack访问Paris&Orléans”，此函数也会返回正确编码的网址。（实际上，在上面的例子中， `iri_to_uri()` 调用不是绝对必要的，因为所有非ASCII字符在第一行的引号中都会被删除。）

数据库API

您可以将Unicode字符串或UTF-8 bytestrings作为参数传递给数据库API中的 `filter()` 方法等。以下两个查询集是相同的：

```
from __future__ import unicode_literals

qs = People.objects.filter(name__contains='Å')
qs = People.objects.filter(name__contains=b'\xc3\x85') # UTF-8 e
ncoding of Å
```

模板

在手动创建模板时，可以使用Unicode或bytestrings：

```
from __future__ import unicode_literals
from django.template import Template
t1 = Template(b'This is a bytestring template.')
t2 = Template('This is a Unicode template.')
```

但是常见的情况是从文件系统读取模板，这产生了一个轻微的复杂性：并非所有文件系统都存储编码为UTF-8的数据。如果模板文件未以UTF-8编码存储，请将 `FILE_CHARSET` 设置设置为磁盘上文件的编码。当Django读取模板文件时，它会将数据从此编码转换为Unicode。（默认情况下， `FILE_CHARSET` 设置为 `'utf-8'` 。）

`DEFAULT_CHARSET` 设置控制着色模板的编码。默认情况下，它设置为UTF-8。

模板标签和过滤器

在编写自己的模板代码和过滤器时要记住的几个提示：

- 始终从模板标签的 `render()` 方法和模板过滤器返回Unicode字符串。
- 在这些地方使用 `force_text()` 优先于 `smart_text()` 。标签渲染和过滤器调用在渲染模板时发生，因此延迟将延迟翻译对象转换为字符串没有任何优势。在这一点上更容易单独使用Unicode字符串。

电子邮件

Django的电子邮件框架（在 `django.core.mail` ）支持Unicode透明。您可以在邮件正文和任何标题中使用Unicode数据。但是，您仍然有义务遵守电子邮件规范的要求，因此，例如，电子邮件地址应该只使用ASCII字符。

以下代码示例演示除电子邮件地址之外的所有内容都可以是非ASCII：

```
from __future__ import unicode_literals
from django.core.mail import EmailMessage

subject = 'My visit to Sør-Trøndelag'
sender = 'Arnbjörg Ráðormsdóttir <arnbjorg@example.com>'
recipients = ['Fred <fred@example.com']
body = '...'
msg = EmailMessage(subject, body, sender, recipients)
msg.attach("Une pièce jointe.pdf", "%PDF-1.4.%...", mimetype="ap
plication/pdf")
msg.send()
```

表单提交

HTML表单提交是一个棘手的领域。不能保证提交将包括编码信息，这意味着框架可能必须猜测提交的数据的编码。

Django采用“惰性”方法来解码表单数据。 `HttpRequest` 对象中的数据只有在访问它时才会被解码。事实上，大多数数据根本没有被解码。只有 `HttpRequest.GET` 和 `HttpRequest.POST` 数据结构具有应用于它们的任何解码。这两个字段将返回其成员作为Unicode数据。 `HttpRequest` 的所有其他属性和方法都与客户端提交的完全相同。

默认情况下， DEFAULT_CHARSET 设置用作表单数据的假设编码。如果您需要针对特定表单更改此设置，可以在 `HttpRequest` 实例上设置 `encoding` 属性。例如：

```
def some_view(request):
    # We know that the data must be encoded as KOI8-R (for some
reason).
    request.encoding = 'koi8-r'
    ...
```

您甚至可以在访问 `request.` GET或 `request.` POST，并且所有后续访问将使用新的编码。

大多数开发人员不需要担心更改表单编码，但这对于与不能控制其编码的传统系统交谈的应用程序是一个有用的功能。

Django不会解码文件上传的数据，因为该数据通常被视为字节集合，而不是字符串。任何自动解码都会改变字节流的含义。